USENIX SOFTWARE MANAGEMENT WORKSHOP

SPRING 1989

U

US

USE

U NIX

USENIX

# WORKSHOP PROCEEDINGS

SOFTWARE MANAGEMENT

April 3 - 4, 1989
New Orleans, Louisiana

For additional copies of these proceedings write
USENIX Association
Suite 215
2560 Ninth Street
Berkeley, CA 94710 USA

The price is $20.

Outside the U.S.A. and Canada, please add
$15 per copy for postage (via air printed matter).

# Program and Table of Contents
# Software Management Workshop
## April 3-4, 1989

## Monday, April 3

*Opening Session* 9:00 - 10:30

Introduction
David Tilbrook

Keynote Speech: Project Hygiene ........ 1
Vic Stenning, Anshar, Ltd.

*Break* 10:30 - 11:00

*Version Management I Chair: Deborah K. Scherrer* 11:00 - 12:30

What a Source Code Control System Should Do
Steven R. Bourne, Sun Microsystems

Software Management using a CASE Environment ........ 11
Masahiro Honda and Terrence Miller, Sun Microsystems

Boxes, Links, and Parallel Trees ........ 17
Andy Glew, Motorola MCD

*Lunch* 12:30 - 2:00

*Version Management II Chair: M. Kirk McKusick* 2:00 - 3:30

The DV System of Source File Management ........ 29
Peter Constantinidis, Jr., and Hamish Reid, UniSoft Corp.

Controlling Software for Multiple Projects ........ 39
Dale Miller, Sterling Software IMD

Transcending Administrative Domains by Automating System Management Tasks in a Large Heterogeneous Environment ........ 51
Melvin J. Pleasants, Jr., Rutgers U., and Eliot Lear, IntelliGenetics, Inc.

*Break* 3:30 - 4:00

*Testing Chair: Barry Shein* 4:00 - 5:30

An Empirical Study of the Reliability of Operating System Utilities .... 59
Barton P. Miller, Lars Fredriksen and Bryan So, U. of Wisconsin, Madison

Panel Discussion
Evi Nemeth, U. of Colorado, Boulder, chair

---

## Tuesday, April 4

*Installation & Configuration Management Chair: D. Tilbrook* 9:00 - 10:30

RAPID: Remote Automated Patch Installation Data Base .................... 77
Russel Brand, Lawrence Livermore Labs., and D. Brent Chapman, Chapman Consulting

Configuration Management of an Environment .................................... 85
Susan A. Dart and Peter Feiler, SEI

CCSLAND ........................................................................................ 87
Nathaniel R. Bronson, III, Multiflow Computer

*Break* 10:30 - 11:00

*Release Engineering Chair: D.K. Scherrer* 11:00 - 12:30

The Release Engineering of 4.3BSD ...................................................... 95
Marshall Kirk McKusick, Michael Karels and Keith Bostic, CSRG

Project Athena's Release Engineering Tricks ....................................... 101
Don Davis, MIT Project Athena

Configuration Management in the X Window System ........................ 107
Jim Fulton, X Consortium, MIT

*Lunch* 12:30 - 2:00

*Miscellaneous Topics: Chair M. Kirk McKusick* 2:00 - 3:30

Automating the Importation of Software ............................................. 115
Paul R. Eggert, UniSys Corporation

The Use of a Time Machine to Control Software .................................. 119
Andrew Hume, AT&T Bell Laboratories

Experiences Using a Hypertext System to Manage Software ................. 125
Peter Nicklin, Hewlett-Packard

*Break* 3:30 - 4:00

*The Grand View Chair: Barry Shein* 4:00 - 5:30

Under Ten Flags (not always smooth sailing) ........................................ 137
David Tilbrook, Nixdorf Computer Canada

Panel Discussion and Wrap-up

---

**Conference Chairs:**

Barry Shein
Encore Computer Co.

bzs@encore.com

David Tilbrook
Nixdorf Computer Canada, Ltd.

attcan!lsuc!gate!t35oa!emsdev!dt

**USENIX Conference Coordinator:**
Judith H. DesHarnais

**USENIX Conference Liaison:**
Deborah K. Scherrer

**Proceedings Production:**
Peter H. Salus
Ellie Young

# Project Hygiene

*Vic Stenning*

Anshar Limited, UK

**Abstract**

It is suggested that many of the difficulties encountered by systems and software projects are not the result of deep technical problems, but rather arise from a lack of basic project hygiene - from failure to enforce various elementary principles and disciplines that are self- evidently prerequisite to a successful outcome. Such disciplines are primarily concerned with the control and co-ordination of both project *activities* and project *products*.

Examination of some of the elementary principles of project hygiene suggests that the principles are often violated. However, while the failings may be obvious, the means for overcoming such failings are somewhat less so. Suggestions are made both on methods and procedures that might improve the standard of hygiene within a typical project, and on topics that demand particular attention.

Of course, simply focusing on hygiene will not of itself ensure project success. The successful development of computer systems and software demands a co-ordinated process incorporating effective methods and supported by effective tools. However, basic hygiene must be at the very heart of the process, and is essential to creating the conditions under which the various methods and tools can usefully be deployed.

Ultimately, project hygiene is rather like any other form of hygiene: nothing spectacular comes from its presence, but the effects of its absence can be dramatic.

## 1. Introduction

Concern here is with basic project hygiene - certain fundamental and obvious principles that would seem to be a prerequisite to any complex cooperative production activity proceeding in an orderly and effective fashion.

Section 2 below briefly discusses some of the more obvious principles of project hygiene. No suggestion is made that the list of principles is exhaustive; it merely serves to illustrate the kind of "common sense" requirements that come under the heading of *hygiene*. However, despite these principles being obvious, it is suggested that many development projects fail to adhere to them. (The term *development* is used both here and elsewhere to encompass not only initial development of some system but also subsequent modification to correct errors or evolve the system's capabilities.) Almost inevitably, failure to enforce hygiene will result either in timescale over-run, or in poor quality products, or both. Not infrequently, lack of basic hygiene leads to complete disaster.

Having identified some basic requirements of project hygiene, section 3 then discusses various ways in which these requirements might be addressed. Unfortunately, while the requirements are obvious, the means of addressing them are somewhat less so, and there is considerable scope for differences of opinion on how the various requirements should best be tackled. However, the important point is that the requirements be recognised and that they be addressed by some means; the specific means employed are of somewhat secondary importance.

Finally, section 4 presents some broad conclusions.

## 2. Some basic principles

A few basic principles of project hygiene are discussed below. These principles are all very simple and very much "common sense". However, in the author's experience, many projects

- including all that end up as major disasters - violate one or more of these obvious principles. Furthermore, while the principles may be self-evident, it is by no means always clear how these principles can be respected, particularly in the case of large projects that require the involvement of a substantial number of people.

**Principle 1**

*Everybody involved in the project should know the objectives of what they are doing*

Or, more bluntly, everybody should know what he or she is trying to do.

This is the fundamental principle of project hygiene, and should not need stating. However, it is surprising how frequently this fundamental principle is violated, if not totally then partially.

Problems of people not knowing what they are trying to do can arise at different levels of the project. At the broadest level, there can be problems with "fuzzy" requirements and with the lack of proper system specifications. These are not in themselves fundamental problems, provided that everyone concerned with the project recognises that the requirements are fuzzy and accepts the implications. The problems arise when the requirements are ill-defined but people proceed as if they were well-defined.

At the intermediate level, teams who are given responsibility for individual sub-systems can misunderstand the role of that sub-system within the whole. It is not unknown for teams to be confused about the functional requirements on their sub-system, and hence to deliver something that simply doesn't do what was expected. However, more commonly the problems at this level relate to non-functional aspects: performance, robustness, fault tolerance, and so on. While there are notable exceptions, it is still generally the case that non-functional requirements on individual sub-systems are inadequately specified, if indeed they are specified at all. The result is individual sub-systems that are under-engineered or - almost as bad - over-engineered. One of the classic symptoms of poor project hygiene is for a team to spend weeks carefully crafting and optimising some component even though, in the context of the system as a whole, the performance of that component is of no real significance.

At the detailed level of the design and implementation of individual modules, problems arise both from communication being too casual and from communication being ambiguous. All too frequently when there are "integration" problems, the cause is not that the faulty component fails to do what its author intended, but rather that the author did not have a sufficiently precise understanding of what was needed. At integration time, comments such as "I thought it meant ..." or "I thought that you were going to ...." or "Nobody told me that ....." are all too common.

These various examples merely serve to illustrate the general point that project personnel often do not fully understand the objectives of what they are doing, particularly in the context of the project as a whole. Given a woefully inadequate specification, which seems to be the norm, the natural tendency is to subconsciously complete the specification in a way that seems consistent and sensible, and then just get on with the job. Often this works. Sometimes it raises problems that have to be fixed. And occasionally it's disastrous.

**Principle 2**

*Achievement of the overall project objectives should follow immediately from achievement of all individual objectives*

This is the necessary complement to principle 1.

Occasionally one encounters projects where everybody seems to know what they are doing, everybody seems to be doing a good job, and yet the project as a whole has major problems. While the causes of such problems can sometimes be very complex, two simple causes seem to recur.

First, there is often a tendency to pay more attention to the sub- division of work than to the subsequent re-combination of the results of that work. This is particularly the case when a project has a tight timescale and the complete team is assigned from day one. Under these circumstances, there is considerable pressure to rapidly partition the work so that each team member has something "constructive" to do. However, given insufficient analysis and design, and insufficient planning for integration and testing, this early partitioning can prove to be more destructive than constructive. The end result can be a "system" that is virtually impossible to integrate and literally impossible to test. More than one system has been completely redesigned during the supposed "integration" phase - another clear symptom of bad hygiene.

Second, in a large and complex project there is a tendency simply to "lose" things. Known requirements simply disappear. Assumptions that are implicit in the specification or design are subsequently overlooked, with the result that the system fails to operate as intended. Work done to explore possible design alternatives or to construct test scenarios is never documented and subsequently has to be repeated. And so on. Traceability is an essential aspect of project hygiene, but is all too easily lost unless effort is consciously invested in its preservation.

### Principle 3

*Both individual objectives and overall project objectives should be realistic*

People tend to be over-optimistic about what they are able to do and, most especially, about how quickly they are able to do it. Equally, people tend to be over-optimistic on other people's behalf. To a typical salesman, most technical jobs are trivial. And, for whatever reason, some senior managers repeatedly make commitments that range from the ambitious to the unachievable. They over-state what they can deliver, or how quickly they can deliver it, or both.

There is no single factor more damaging to project hygiene than unrealistic targets. By definition, a project with unreasonable objectives starts badly, but there is subsequently a very real danger of positive feedback, taking things from bad to worse. When trying to achieve the unachievable there is an obvious temptation to cut corners, to experiment with short cuts, and to avoid doing anything that does not address an immediate and urgent need. The end result is usually a complete loss of hygiene, leading to the project taking far longer than it would have done had it been properly planned and controlled, and delivering far worse products.

Of course, goals that were initially reasonable can either emerge as, or become, unrealistic as the project proceeds. Time can be lost to unforeseen problems or to changing requirements. Thus, basic hygiene demands regular monitoring of project status, not just in the narrow sense of progress against plan, but in the broader sense of whether the overall goals and strategy remain reasonable. Such a judgement cannot sensibly be made in isolation, considering only the ultimate goals, but rather must be based on a realistic assessment of the project's current status.

### Principle 4

*There should be a known method for addressing each individual objective*

Employing appropriate methods is an essential aspect of project hygiene - reliance on personal inspiration or hidden magic is unhygienic by definition.

Insistence on the use of known methods is not the same as insisting that we must know in advance a way of solving every unknown problem. When a new problem is encountered, we may well not know the method by which that problem should best be tackled. But under these circumstances we should know how we will explore the problem and determine the method by which the problem should be addressed - that is, we should have a method for selecting methods. In general, the methods that are employed at any step of the project may be governed by the results of previous steps, rather than prescribed in advance.

Thus, project hygiene does not demand the existence of some universal or infallible method, nor does it prohibit the use of experimental or exploratory methods. It simply requires that in addition to knowing what we are trying to do, we also know how we are trying to do it.

## Principle 5

*Changes should be controlled, visible and of known scope*

In software projects, change is the norm. Even in a project that is developing a completely new system, only a small proportion of the effort is devoted to original creation. The major part of the effort is concerned with modification of something that already exists, for example when performing a design iteration or when correcting program errors.

The fact that change is continuous makes it difficult to preserve consistency between the project's various products. The problem is exacerbated by the fact that frequently a large number of changes are in progress simultaneously. Changes arise from a number of different sources - changing requirements, detected errors, new design decisions, problem reports, and so on - and typically such changes cannot be handled serially. The interactions between the different changes, and the impact on the different variants of the system, then become extremely difficult to co-ordinate.

Inadequate change control is one of the most common forms of poor hygiene, and its symptoms are all too familiar. "Minor" changes have an impact that was totally unanticipated. The ramifications of a change are not followed through, so that modules affected by the change are not properly updated. Incompatible changes are introduced simultaneously. And so on.

Several of the fundamental hygiene requirements are obvious. For any proposed change, it should be possible to assess the impact of the change, and the possible scope of its effects, before the change is made. Where changes potentially "interfere", they should either be serialised or merged into one combined change that can be properly controlled. Those whose work will be impacted by the change should receive prior warning before the change is made, rather than simply seeing its results.

However, while such basic requirements are easily identified, they are unfortunately hard to satisfy.

## Principle 6

*Both people and products should be insulated from the effects of changes that are not (currently) of relevance*

In an environment of constant change, it is necessary for individual activities to be insulated from the impacts of changes until they are ready to receive them. While an individual activity is in progress, it should proceed in a stable context.

This is not to say that any extant activity must always proceed to completion, irrespective of any changes that arise while that activity is in progress. Obviously such a constraint would be both undesirable and impractical. However, where an extant activity is to be impacted by some change, this should be explicitly recognised and explicitly co- ordinated. The specification of what the activity is to do should be updated appropriately, and the implications of this update taken into account by the project planning, monitoring and co-ordination functions.

Lack of proper control in this area usually manifests itself in two ways. Firstly, both activities and products are forced to accept changes at inopportune times, simply because there is no mechanism whereby the acceptance of such changes can be postponed. Second, activities are continually subject to externally-imposed change, but all such changes are introduced implicitly, and the project co-ordination function treats the activities as if they are proceeding normally in a stable context. Both classic examples of poor hygiene.

## 3. Means of improvement

Section 2 above outlined a few of the basic requirements of project hygiene, and also indicated that projects often fail to meet these requirements. Unfortunately, identification of the obvious problems is rather easier than identification of solutions. However, without in any way claiming to offer such solutions, this section suggests some basic ways in which hygiene can be promoted and the hygiene level of the "typical" project improved.

### Suggestion 1

*Focus on the process as a whole, rather than on the (final) product*

Despite the considerable progress over the past several years, our industry still seems to over-emphasise source code and under-emphasise almost everything else. There still tends to be a suspicion that a project hasn't properly started until at least some code is actually running. We still tend to regard "software management" as being synonymous with "source management", when really the topic should be far richer.

Hygiene is not an attribute of an end product, but rather is an attribute of the process by which that product is produced. To improve project hygiene, we must focus on this process. Furthermore, we must avoid the temptation to concentrate on some particularly interesting aspect of that process at the expense of other, duller aspects: hygiene comes more from achieving overall co-ordination of the various process components than from improving individual components in isolation.

Four major aspects of the process that must be co-ordinated are project management, technical development, configuration management, and quality assurance. Frequently, these different aspects are treated as separable elements, and each is addressed largely in isolation of the others. As a direct consequence, the process as a whole lacks coherence. Managers have little visibility either of intermediate technical objectives or of technical progress, and therefore have no basis on which to manage the project effectively. The configuration management procedures are more symbolic than effective: they apply only to the code, hinder desirable changes, and fail to prevent undesirable changes or to promote proper change co- ordination. And, in the absence of effective project management and effective configuration management, the quality assurance function is severely impaired. In short, many people involved in the project will be unable to do their jobs effectively, and some will not even know what they are trying to do.

Emphasising the co-ordination of the many different aspects of the process is a good first step in promoting improved hygiene. However, it should be recognised that the source code provides a very poor basis for such process co-ordination. This leads immediately to the second suggestion.

### Suggestion 2

*Invest more effort in "higher level" descriptions*

Far too many project still rush into code. Any time spent on higher level descriptions of the system - requirements, specifications, designs - is spent grudgingly and sparingly. Sometimes such descriptions are not produced at all. Sometimes they are produced after the event, documenting the implementation to meet some contractual commitment, rather than as an integral part of orderly development.

Unfortunately, the distance between some vague notion of what we want and actual running code is enormous. If the entire project basically takes the form of a single gigantic leap from one to the other, it is difficult to imagine how any reasonable level of hygiene is ever to be achieved.

On any non-trivial project it seems essential to employ a sequence of higher-level descriptions to bridge the gulf between concepts and code in a series of manageable steps. The number and nature of these descriptions may vary dependent upon the application, but the sequence in some form will always exist. The technical objective of the project is not then to produce the final code, but rather to produce this complete harmonious sequence, together with related information on design decisions, verification exercises, and so on.

Of course, some forms of higher level descriptions are already employed fairly extensively: natural language specifications and data flow diagrams, for example. However, project technical personnel often take a rather cynical view of such descriptions. If the code is working, and the code doesn't correspond to the diagram, then it's the diagram that's wrong. When an update is required, the code gets changed but the diagram doesn't, leading eventually to the diagram becoming completely obsolete. The attitude is very much that it's the code that really matters, and the rest is so much arm waving.

At least part of the reason for the preoccupation with code is that the code has complete and precise semantics, which is in sharp contrast to most forms of higher level description in common use. There is enormous value in precise semantics: we can determine exactly what the system does in a given set of circumstances, with no doubt or ambiguity. Unfortunately, in the case of code, this precise semantics is at a level of detail which is very difficult for people to understand and for either people or tools to manipulate.

The failing lies neither with the code, nor with our desire for precise semantics, but rather with the imprecision of our higher level notations. This failing can be addressed in various ways. We can improve the precision with which we use our established notations, usually by being more precise in our use of natural language. We can switch to related notations with richer and more precise semantics. But ultimately, to completely overcome the failing, we need to adopt formal notations with semantics as precise as those of our programming languages.

Of course, there are many objections to the use of formal notations: they are very difficult and expensive to use, they don't ultimately ensure high quality results, they lack flexibility and generality, and so on. Some of these objections have some basis, others are groundless. What seems clear is that project hygiene benefits enormously from - and perhaps even demands - precise higher-level descriptions of the system under development. With informal notations, even vaguely approaching the level of precision required demands huge effort. Formal notations are the only known solution to the problem.

Of course, once the idea of formal notations has been accepted, there is scope for further improvement with the introduction of rigorous development and proof, typically on a selective basis. However, one step at a time: the initial requirement is to employ higher level descriptions and to make these descriptions as precise as is possible in the context of the particular project.

**Suggestion 3**

*Co-ordinate activities as well as products*

As has already been discussed, there has been a traditional tendency in the software industry to focus more on products than on process, and particularly on the program code.

Presumably as a consequence of this tendency, most of the co- ordination mechanisms that have been developed have been concerned with the co-ordination of products. We have change control mechanisms, general configuration control mechanisms, interface checking mechanisms, and so on. But there is a noticeable lack of mechanisms for co-ordinating project activities: the few mechanisms that are available seem to be adjuncts to product co-ordination systems, rather than specifically directed at the co- ordination of activities.

There is an obvious duality between activities and (intermediate and final) products. One can either view a project as consisting of a product structure that is manipulated by activities, or one can view it as consisting of a set of activities that "produce" and "consume" products. Ideally, consistency between the two views should be enforced, and it should be possible to switch between the two as appropriate. Certainly the typical current situation - where the activity view is the province of the project management function and the product view is the province of the configuration management function, and never the two shall meet - is highly undesirable.

Recognition of the activity/product duality suggests the use of a common model encompassing both. The individual views can then be obtained by appropriate projections from this common model. Such an arrangement would promote basic hygiene, in that a minimum level of consistency would be guaranteed. Potentially, such a common model could also provide a foundation for configuration management, as discussed under "suggestion 5" below.

**Suggestion 4**

*Adopt a strict policy on project phases*

Section 2 discussed the problem of project hygiene being completely destroyed by unrealistic objectives. Unfortunately, it is not always possible at the outset of a project to judge the realism of the objectives, simply because there is insufficient information on which to base such a judgement.

Under these circumstances, where the full scope and difficulty of the project are unknown, hygiene can be promoted by sub-dividing the project into self-contained phases - so self-contained that each phase is, in effect, a separate project. While the overall objectives and projected timescale may be known in outline, only the objectives and timescale for the current phase will be defined in detail. The objectives and timescales for the next phase will then be defined towards the end of the current phase. And, dependent upon knowledge gained during the current phase, the overall objectives and projected timescale can always be modified - perhaps drastically.

Each phase will typically produce the next description in the sequence of higher level descriptions discussed earlier. Each phase will identify new problems and generate new knowledge, and thus will provide the basis for a proper definition of the next phase. Hygiene can be preserved, because the project team never needs to commit to solving an unknown problem.

Of course, such an approach requires not just the support of the project team, but also the co-operation of the "customers". While the latter may initially be reluctant to accept what might be seen as an open-ended commitment, they can usually be persuaded that the approach is in their own best interest - which indeed it is.

**Suggestion 5**

*Provide more semantic information to the configuration management and build systems*

As mentioned in section 2, the co-ordination of continuous and concurrent changes is one of the most difficult aspects of project hygiene. While many mechanisms and tools have been developed to address various aspects of this problem, and many of these have been extremely valuable, there are still no complete solutions available. Indeed, given the nature of the problems, complete solutions are difficult even to imagine: one can only hope for steady improvement, both in terms of the range of problems that are addressed and in the effectiveness with which they are addressed.

Of necessity, in order to provide general and accessible solutions to urgent problems, most of the available change control mechanisms are based upon very simple models. The fact that something has changed may be measured, and the direct impact of that change assessed, by the use of date/time stamps. Concurrent changes to individual modules are restricted by the use of check-out/check-in mechanisms. And so on.

Clearly such mechanisms perform a useful function, but equally clearly they address only part of the problem. Assessing changes on the basis of date/time stamps can become extremely inconvenient when one wishes to make a limited change to some previous release of the system. Check-out mechanisms alone do not address the issues of maintaining a consistent set of modules when a change to one module may impact several others.

Over the past several years, there has been considerable progress in the field of change management -in the context of such languages as Mesa, Modula and Ada - from exploiting the syntactic structure of inter-component interfaces. Several of the more trivial, and more common, problems of change can be eradicated in an environment where syntactic interface checking is enforced.

There is considerable scope for further improvement by extending the checking to encompass not just syntax, but also semantics. Such extension of course requires precise formal specification of the kind discussed earlier. By performing semantic checks it would be possible to ensure that a component module was functionally compatible with the remainder of the system or, in the case where it is not, to indicate the other components that are impacted.

Like other aspects of the overall process discussed earlier, the field of change control and configuration management has suffered from an undue preoccupation with source code. Fortunately, there is now widespread recognition that configuration management of higher level descriptions, of user documentation, of test histories, and so on, is just as important as configuration management of code.

Eventually one would like to see complete project/product graphs that encompass all the intermediate and final products with which the project is concerned, their components, all the versions and variants, and the semantic relationships between them. Such a graph would obviously be both large and complex, and would require some form of automated consistency preservation system. Some of the relationships would initially be asserted by members of the project team, and as products begin to emerge they would be checked for consistency with those assertions. Other relationships would be derived from the products themselves. The impact of proposed or actual changes, to either a higher level description or a code module, would be assessed both syntactically and semantically. The mechanisms would be able to report the impact of the change on all products of interest, including "old" variants.

Such a system, which could make a huge contribution to this very important aspect of project hygiene, may still be some way off. But steady progress is being made in this general direction, and we have grounds for optimism.

## 4. Final remarks

One of the reasons for poor hygiene being so prevalent is that people tend to become so engrossed in the technical demands and details of what they are doing that they lose sight of the broader picture. Not infrequently, opportunities for improving basic hygiene can be identified simply by standing back from the day-to-day business of the project and asking a few fundamental questions of the kind suggested above; for example

- are the objectives of all project activities clearly and precisely defined?
- are there any activities for which the objectives are completely unrealistic?
- how will the results of the individual activities be combined to meet the objectives of the overall project?
- are there explicit procedures and mechanisms to ensure traceability?
- are there effective mechanisms for co-ordinating changes and limiting the scope of their impact?

Sometimes, when the answers to such questions are not entirely satisfactory, one can readily identify obvious minor enhancements to the process being employed that will yield some rapid improvement. Often, the purpose of such enhancements is to make explicit some aspect of the overall process that was previously handled implicitly. For example, it may be appropriate to require explicit specification of objectives for all individual project activities; or to introduce a new procedure for authorising changes to some level of description; or to insist that specifications encompass non-functional characteristics as well as functional ones; or to instigate a formal "internal" release mechanism to complement the external one. Anything that will promote awareness of the real objectives, stability in the face of change, and realistic assessment of the true position, can only enhance hygiene. Such modest changes may not entirely solve the perceived problems, but they can yield immediate benefit.

Considerable progress can often be made by insisting that what was previously implicit and assumed be made explicit, by introducing basic procedures for co-ordination and communication where previously none existed, and by extending the scope of established QA review procedures. All this can be done at the "process glue" level, without changing the individual technical and management methods that are employed within the process.

However, regardless of the position from which one starts, and regardless of the route by which progress is made, the need will eventually emerge to make higher-level descriptions more precise. At some stage, further progress with project hygiene will require the introduction of formal specification and design notations with precise semantics.

Probably the most difficult area of project hygiene is the co-ordination and control of the many versions of the various intermediate and final products, the components of those products and their relationships. While much valuable progress has been made with mechanisms to address various individual aspects of this huge problem, there has been little sign of a coherent mechanism to address the problem in its entirety. The "project/product graphs" envisaged in section 4 above could offer one route to a solution, though perhaps not the only one or the best one. Of all the research topics that could usefully be addressed under the broad heading of project hygiene, this is probably the most urgent and the most important.

# Software Management Using a CASE Environment

Masahiro Honda and Terrence Miller

Sun Microsystems, Inc.

2550 Garcia Ave

Mt. View, CA. 94043

({honda,tcm}@sun.com)

The Sun Network Software Environment (NSE) is a network-based object manager for software development and computer–aided engineering. The NSE supports parallel development through an optimistic concurrency control mechanism, in which developers do not acquire locks before modifying objects. Instead, developers copy objects, modify the copies, and merge the modified objects with the originals. The NSE is designed to support both developers and release engineers who are working on large software projects.

## 1 Introduction

The creation of a large software product is an exercise in programming-in-the-many:

1) many objects

2) many people

3) many releases

4) many machines

The following sections describe how these areas are addressed by the Network Software Environment (NSE) developed by Sun Microsystems. The NSE is a network-based object manager designed to support closely cooperative, parallel development work by moderate sized (20 people or less) work groups and provide configuration management for larger efforts. A more detailed description of the NSE is presented in [1].

## 2 Compound Objects

Large software projects involve many objects. Developers must deal with sets of objects as units. They must be able to identify the versions of *all* the objects that make up a consistent unit, such as a release. The NSE uses compound objects to facilitate the management of large numbers of objects.

### 2.1 Components

The NSE operates on sets of objects called *components*. Components can be used to group objects together so that they can be managed as a single unit. An NSE component is a general-purpose compound object that can be used to group any collection of related objects. Objects of any type can be members of a component, including other components, and one object can be a member of two or more components. Although the NSE does not restrict the contents of components, a typical component might contain a target object representing a program, and all other objects related to that program.

Because components can contain other components, they can represent the hierarchical structure of a complex software product. A typical system might consist of one top-level component representing the entire system; this component might have one component per subsystem, and these components might have subcomponents for programs. When components are used in this way to represent levels of abstraction, each component can contain an intellectually manageable number of objects, say, five to ten. Some of these objects will be components representing the next lower level of detail. To see the next level of detail you can examine the subcomponents.

### 2.2 Targets

An NSE *target* is a compound object which dynamically computes its set of sub-objects. It automates much of the bookkeeping associated with derived files generated using the `make` utility [2]. The members of a target are a derived file and the objects needed to build it. The objects needed to build a derived file consist of a *Makefile*, and a collection of *dependencies*. A Makefile is basically a recipe that describes how to construct a derived file in the fewest possible steps. Dependencies are files such as source files, object files, libraries, and header files, that, if changed, require the derived file to be rebuilt. The NSE automatically keeps the contents of targets up to date as Makefiles and dependencies are changed.

The NSE does not require that users to explicitly list all dependencies in the Makefile. For example, it automatically includes files referenced in `#include` statements in C programs.

## 3 Copy-Modify-Merge

Large software projects involve many people working together on a common set of objects. Problems arise when more than one person must update the same object concurrently. NSE object management is based on a *copy-modify-merge* paradigm. Simply stated, a developer copies an object without locking it, modifies the copy, and then merges the modified copy with the original. This paradigm allows developers to work in isolation from one another since changes are made to copies of objects. Because locks are not used, development is not serialized and can proceed in parallel. Developers, however, must merge objects after the changes have been made. In particular, a developer must resolve conflicts when the same object has been modified by someone else.

Our design is related to that of the Cedar System Modeller [3]. Cedar provides mechanisms for defining sets of consistent files. Developers copy a set of files to their workstations and modify the copies in isolation. The modified copies are then copied back to be merged with the originals. Cedar, however, does not have environments (see next section).

## 3.1 Acquire

Since software development involves multiple objects, it is not sufficient to just make copies of the objects that are to be modified. Instead, the entire set of objects that make up a program or system must be copied as a unit, so that after the changes are made, the changes can be tested to make sure they are consistent with all objects in the set. Developers must therefore be able to locate and copy consistent versions of objects on which to apply their changes. The NSE implements the copy step of the copy-modify-merge paradigm through an operation called `acquire`. `Acquire` takes a component as an argument and creates a (logical) copy of the versions of objects in the component. Consistent versions of all related objects are therefore copied together.

The `acquire` operation normally does not create physical copies of file objects. Instead, the file is shared until the copy is modified. The extensions to the Sun filesystem which implement file sharing and copy-on-write are described in [4].

Objects are acquired between NSE *environments*. An environment is a work space that contains (logical) copies of objects. Each environment can have a different copy of an object. An environment also provides a virtual name space for its objects, so the same name can be mapped to different versions of the object depending on which environment the name is referenced in. The apparent names for objects do not change when they are copied. Thus references made from one object to another, such as absolute references to include files from source files, do not need to be changed.

A developer acquires a component from a common *parent* environment to a private *child* environment. The parent environment serves as an integration area for a group of developers working on the same project. Each developer updates acquired objects in his or her child environment in isolation.

## 3.2 Activate

In order to work on the objects acquired into an environment, the user activates the environment. Within the context of the `activate` command, a portion of the file hierarchy is replaced by the files and directories contained in the environment and all tools which read and write simple files access the files appropriate to the activated environment. A similar functionality is provided by the Configuration Treads of DSEE [5] but the NSE preserves full, hierarchical pathnames. The per-process directory mounting techniques used to implement activation are described in [4].

Within an activated environment the NSE provides file checkout and checkin facilities to capture change comments and protect against accidents. Unlike SCCS [6] or RCS [7], the NSE's version control facility does not allow branching. Environments are the units of parallelism and are not themselves used in parallel by more than one developer.

## 3.3 Preserve

The `preserve` command saves snapshots of components to create *components revisions*, so that consistent versions of objects can also be grouped together as a unit. A revision of a component in one environment may be later acquired into a new environment in order to resume development from that point.

## 3.4 Reconcile

After making changes and testing them in the child environment, a developer reconciles the modified component with the parent. The `reconcile` operation copies the changed objects in the component back to the parent environment. The NSE keeps track of what objects have been changed in an environment. This information is used to determine what objects to copy back and also to determine what objects are in conflict as a result of parallel changes.

A `reconcile` by a developer will fail if any object contained in the component has changed in the parent since it was acquired. This usually means that a second developer in another child environment has concurrently changed and reconciled objects that are in the first developer's component. The `reconcile` fails because the first developer's changes may no longer be consistent with the second developer's changes. For example, the second developer may have changed a common include file, which could cause the first developer's changes to no longer work.

## 3.5 Resync

When `reconcile` fails. it does not copy back the changed objects. Instead, it calls the `resync` operation to acquire into the first developer's environment the new versions of objects in the parent. If an object has been changed only in the parent, the new version replaces the old one in the child. If the object has been changed in both parent and child, a conflict exists. `Resync` acquires the information needed to resolve conflicts; this is usually a copy of the new version from the parent and the version which is the common ancestor of the versions in parent and child.

### 3.6 Resolve

The NSE `resolve` operation can be called after resync to step through each conflicting object, and to invoke the appropriate tool to resolve conflicts on each object depending on the object's type.

For ASCII files, the NSE provides a window-based merging tool, called `fileresolve`. Fileresolve uses information from the common ancestor of the two version being merged to facilitate the merge.

## 4 Release Environments

Large software projects involve development and maintenance of many releases at the same time, which requires the same object to have many versions undergoing change. Problems arise in trying to merge changes made in one release with changes made in another release. We have seen three patterns in the use of NSE environments which address different problems related to software releases.

If a release environment is initially created as an empty grandparent of the development environments, the act of reconciling a component from the integration environment determines public visibility of a set of changes. A single person can coordinate testing in the integration environment after locking it against further reconciles by developers and verify that the component is ready for release.

By putting releases in separate environments, `resync` can also be used to update a new release with changes, such as bug fixes, made in an old release. When work on a new release is to start, an environment for the new release is initialized by acquiring objects from the old release environment. Changes can then occur in parallel to both releases. As bug fixes are made to the old release, the changes can be resync-ed to the new release.

Minor releases can be represented as revisions of the release environment. If it is necessary to work with a previous minor release, it can be acquired into a new environment. After any required changes have been made and patches generated, the new environment can be resync-ed with the latest versions from the release environment. After resolving conflicts, the result of merging the emergency fixes can be reconciled back into the release environment.

## 5 Distribution

Environments have network-wide names, and can be accessed from any machine on a network. Developers need not know the physical network topology in order to access environments.

The NSE programs which perform `acquire` or `reconcile` never access the remote environment directly (except when sharing files via NFS access which is optional). They communicate via Remote Procedure Call (RPC) with a server running on the machine at which the environment is located. Thus the NSE design allows `acquire/resync/reconcile` to work between environments with widely varying implementations. Except during those operations, there is no communication between environments (unless files are being shared) and local work can continue even if the communications channel is interrupted.

## 6 Conclusion

The NSE is a CASE environment which attempts to address many of the problems of software management. It has been used to support its own development and has been released as product.

## 7 Acknowledgments

This paper reflects the work of the entire NSE team at Sun Microsystems. People involved included Evan Adams, Jonathan Feiber, Jill Foley, David Hendricks, Tom Lyon, Russell Sandberg, and Daniel Scales.

## 8 References

[1] W. Courington, *The Network Software Environment*, Sun Technical Report FE197-0, Sun Microsystems Inc, February, 1989.

[2] S. I. Feldman, "Make – A Program for Maintaining Computer Programs", *Software: Practice and Experience*, April 1979.

[3] B. Lampson and E. Schmidt, "Organizing Software in a Distributed Environment", *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, June 1983, pp. 1–13.

[4] D. Hendricks, "The Translucent File Service", *Proceedings of the European Unix Systems User Group Autumn 1988 Conference*, October 1988.

[5] D. B. Leblang and R. P. Chase, "Computer-Aided Software Engineering in a Distributed Workstation Environment", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering on Practical Software Development Environments*, April 1984, pp. 104–112.

[6] Bell Telephone Laboratories, "Source Code Control System User's Guide", *UNIX System III Programmer's Manual*, Oct. 1981.

[7] W. F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System", *Proceedings of the 6th International Conference on Software Engineering*, Sept. 1982.

# Boxes, Links, and Parallel Trees: Elements of a Configuration Management System.

*Andy Glew*
*aglew@urbana.mcd.mot.com*

Motorola Microcomputer Division, Urbana Design Center
1101 E. University, Urbana, IL 61801

*ABSTRACT*

There are two widely used source code control systems for UNIX®[1]: SCCS and RCS. There have been many *wrappers* placed around these systems to make them more useable for large projects, such as Sequent's Software Change Control System [MosChri86]. There are also many source code control systems that started from scratch, such as Paisley's SCM [MBCHJN86], or Yost's Rich Person's Revision Control System [Yost85]

At "The Little Software House on the Prairie", we devised yet another set of wrappers around RCS to meet the needs of large projects doing development on multiple machines sharing large amounts of code. These wrappers evolved to combine three fundamental features, *parallel trees*, *boxes*, and *links*, which in their combination made for a powerful, convenient, and economical source code control system. Our system also had features to support bug tracking and semi-automatic testing of modules. These features were important and made managers happy; but boxes, links, and parallel trees made possible a system that developers could enjoy using. We hope that future configuration management systems will have these features.

## 1. INTRODUCTION:

**1.1 Ideas, not a Package:** This paper presents ideas that were in several software configuration management (CM) toolsets created at our site, "The Little Software House on the Prairie".[2] These ideas were implemented in various toolsets, with varying degrees of success, and after experience with these toolsets we feel that the following three ideas, *parallel trees*, *boxes*, and *links*, are worth keeping around. We hope to see these ideas in future CM tools.

This isn't an academic paper, although I do give in to my tendency to try to think about things in an abstract way; and it isn't a paper that says "you should do things this way"; if anything, I am trying to write a paper that says "we have found these ideas useful, you may too", much as we found [Yost85] and [MosChri86] sources of practical, possible, ideas for CM.

## 2. BACKGROUND:

**2.1 Alphabet Soup:** As mentioned above, our site experimented with a succession of CM toolsets. In the dim past these were wrappers around SCCS; more recently, they have been wrappers around RCS.

A brief history of these toolsets may be useful to understanding the evolution of some of these ideas. The features will be explained in subsequent sections.

| | |
|---|---|
| Ktools | "K" for kernel. Used for kernel development on a large project. Featured symlink sharing of files; integrated testing. |
| Btools | "B" for box. Used for complete system development on a small project, also for maintenance of a benchmark suite, and personal stuff. Featured environment variable based name transformation, parallel trees, boxes for isolation, link sharing for space |

1. UNIX and System V are trademarks of AT&T. UTX/32 and UTX/32S are trademarks of Gould. SYSTEM V/68 is a trademark of Motorola.

2. Formerly Gould Computer Systems Division's, now Motorola Microcomputer Division's, Urbana Design Center. The tools described here are not official products.

| | |
|---|---|
| | saving, locks, and consistency checks. |
| Gtools | "G" for generic. A student project evolving out of the Ktools and Btools, concentrating on improving name transformations to make life more convenient for users of the system, and improved reliability. |
| Utools | "U" for UTX. Used for complete system development between two large projects. Featured parallel trees, multiple overlaid source trees, shared code between projects, boxes, and link sharing. Coded in C for greater speed and reliability than the earlier shell scripts. Slightly less sophisticated name handling, less convenient for users. |
| KUtools | "KU" for kernel and UTX. Integrated test features with the Utools, as in the earlier Ktools. |

This "alphabet soup" of toolsets spanned about three years. Evolution was slow because of the inertia of commercial software development: a toolset will only be changed at the beginning of a project, not as schedules get tight towards the end.

This line of evolution ends with Motorola's purchase of the site like all companies, Motorola has their own internal CM toolset:

| | |
|---|---|
| Ptools | "P" for project. SCCS based. A slightly different flavour of parallel SCCS trees; no real concept of source code views. No boxes, no links, although these can be added. Less flexible name transformation. |

Sometimes compatibility is more important than flexibility, so we are adopting the Ptools as our standard CM toolset. This is a good time to record for posterity the good ideas in our RCS based CM toolsets.

**2.2 Large Projects with Much in Common:** Raw RCS (or SCCS) are sufficient for configuration management of small projects with limited numbers of developers. However, they don't work all that well with large projects[3].

At times, all of our site was basically one large project, with 50-100 people working on the same product simultaneously. Actually, there were multiple projects, but all were developing varieties of UTX, Gould's 4.x BSD based OS, tracking SUN and AT&T System V features as well. There were two main processor families, with completely different instruction sets, memory and I/O systems, necessitating different kernels. There were also varieties: "Standard" UTX, and various value-added flavours like Real-Time UTX and Secure UTX. Plus there was maintenance on already released products, as well as development on new releases.

Thus, there were several different kernels, but much code was common to all projects. User level code, in particular, was shared, as was high level kernel code such as networking. Our CM system evolved to facilitate the sharing of code between systems, to ensure that bug fixes made to one product got into all products, and to reduce inconsistencies between product lines. Such inconsistencies can be annoying when moving between different architectures from the same manufacturer.

This sharing allowed projects to be smaller. Since all projects had to deliver a full UTX product (even if only an add-on package, they needed to control the rest of the environment to prevent compatibility problems), the projects would otherwise have had to devote people to tracking changes in other systems. But there is no free lunch: teams making changes to the shared code had to do more work, ensuring that their changes worked on all systems, not just the project they were working on. Such testing, of course, is not the type of work that most programmers enjoy.

3. 30-80 people is really only a moderately sized project by the standards of the rest of the computer industry, but is fairly large for a UNIX operating system shop.

## 3. PARALLEL TREES:

The idea of putting things in parallel directory trees seems obvious to many people, and obscure to others. Once you have decided to put things in parallel trees, you must have operations to move between the trees; this *name transformation* is an implementation detail that greatly influences the ease of use of the system, so I discuss it in some detail. Forgive me if you find this section boring.

**3.1 Parallel Source and RCS trees:** In raw RCS there are two files: the source file, and the *history* in the RCS file. When you have multiple developers working on the same file you have multiple versions of the source file, and a single RCS history file. Without the RCS files the source files are organized into a large directory tree. The question is, where do you place the RCS files?

Rather than placing the RCS files next to each of the source files in a single directory tree, we created a directory tree structure completely parallel to the source tree, and placed the RCS files there. This gives us *parallel trees*, with RCS files in the RCS tree, and source files in the source tree (SRC). The basic operation is a mapping between these two trees, path *name transformation*.

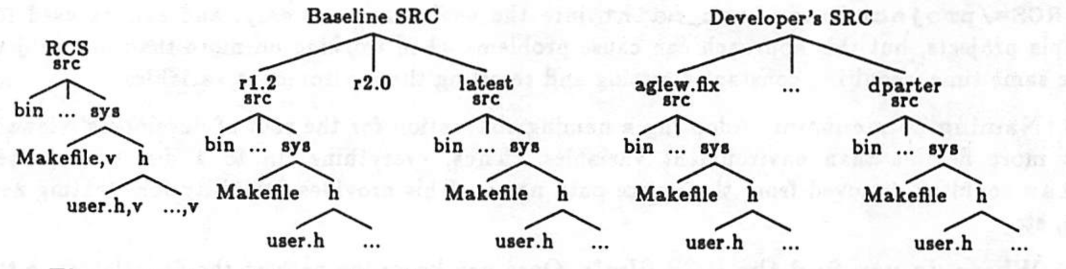


**Figure 1.** Parallel trees for different purposes

The fundamental advantage of separating the SRC and RCS files into parallel trees is that we have formed a single *history object* for version control. The parallel tree is actually a rather primitive history object; it does not perform revision control on files themselves, as opposed to the data within the file - eg. it does not provide revision control for file permissions, file deletions, etc., unless you encode these within scripts - but it is a beginning. CM systems should go further towards making comprehensive history objects.

**3.2 Alternative: Symlinked RCS directories:** We did try some alternatives to parallel trees. The original Ktools had RCS directories in every directory of every developers' source trees, shared by symlinks (Figure 2). To find the RCS file corresponding to a source file, all you had to do was look in the RCS subdirectory next to the file, instead of having to walk up, sideways, and down in the parallel trees.

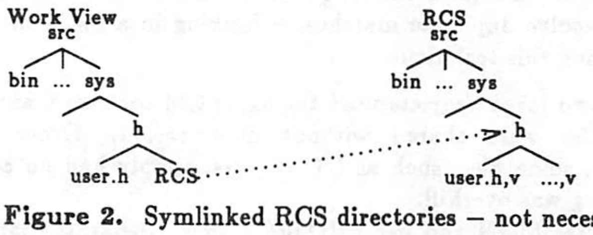


**Figure 2.** Symlinked RCS directories – not necessarily parallel trees

Symlinks work, but interfere with the use of `chroot` boxes; eventually, we discarded symlinks in favour of parallel trees with hard link sharing.

**3.3 Name transformations:** With parallel trees a source file and its corresponding RCS file might be separated by a long distance. Fortunately, RCS allows independent specification of both source file and RCS file on the command line.

This was an important human-interface consideration; no developer wants to type out two extremely long filenames, like

```
ci  /project/base/src_maint/src/sys/h/user.h,v
    /work/maint/aglew.view/src_maint/src/sys/h/user.h
```

Relative pathnames may help abbreviate one name, but not both, since they exist in two parallel trees.

The name transformation is simply, given the absolute pathname of the source file (easily obtained, although special consideration must be provided for symlinks and upward relative components), delete the prefix of the developer's work view `/work/maint/aglew.view` and substitute the prefix of the RCS tree `/project/base/src_maint`. Several ways of doing this are discussed briefly in the next sections.

**3.3.1 Wired In:** One easy way to do this is to wire the paths into the toolset; embarassingly, that is what we finally settled on with the Utools, because it provided one less way for the tools to break. Although we tried many other approaches all had their problems.

**3.3.2 Environment variables:** Placing the prefixes `BOX=/work/maint/aglew.view` and `RCS=/project/base/src_maint` into the environment is easy, and can be used for multiple projects, but this approach can cause problems when working on more than one project at the same time, requiring constantly setting and resetting the environment variables.

**3.3.3 Naming convention:** Adopting a naming convention for the root of developers' views is a bit more flexible than environment variables. Thus, everything up to a directory named `*.view` could be removed from the source path name. This provides flexibility for creating new views, etc.

**3.3.4 Where do you find the RCS files?:** Once you know the path of the file relative to the root of the source tree (developer's view) you still need to know where the RCS tree lives. In a situation where a developer is only working on one or a few projects at a time, wiring the path in or setting an environment variable is ok. But if a developer is working on several things which are under different CM systems it would be nice to have a better default way of determining where the RCS tree is.

We examined placing these in a file at the root of the view (determined by the naming convention above, or by an environment variable). This works, until you come to a situation where not only are there many possible source files for each RCS file, but also where there may be multiple RCS files for each SRC file (which happens as two projects with similar but separate code bases converge).

**3.3.5 Database Driven:** Motorola's Ptools do name transformation by prepending an environment variable `PROJSRC` to the pathname they are given (without even generating absolute paths); failing this they search through a Bill of Materials for a file of the same name, and interactively query the user to resolve duplicate matches. Checking in a file with a common name, eg. `Makefile`, is onerous using this technique.

**3.4 Overlaid Trees:** Eventually two large projects used the same CM tools to manage a large amount of shared code. Many files were shared without modification. Other files were `#ifdef`'ed appropriately. However, some files, such as I/O drivers, simply had no counterpart on the other system, and `#ifdef`'ing was overkill.

The source files were split into COMMON and per PROJECT files. Separate, parallel, RCS trees were created for the COMMON files, and for both projects' private files. Logically, these trees overlaid each other to create two complete, but overlapping, history objects, as in Figure 3.

This could be generalized to tree paths; but that goes counter to the goal of sharing as much as possible.

**3.5 Views:** It has already been mentioned that each developer would have her own *view* of the source tree. This would actually be a separate parallel tree, from the root on down. Files that the developer did not need weren't copied. This led to a lot of strange-looking nearly empty

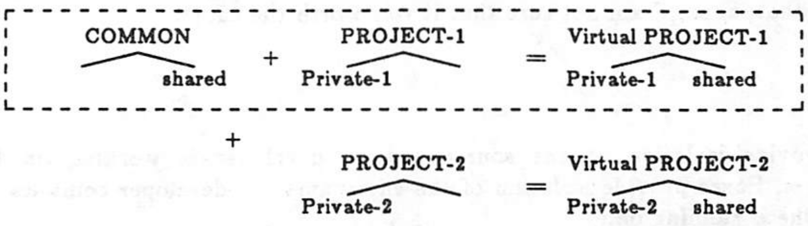


**Figure 3.** How COMMON and PROJECT trees are overlaid

directories (Figure 4), but was convenient for name transformation, and when you needed to pull something forgotten into your view.

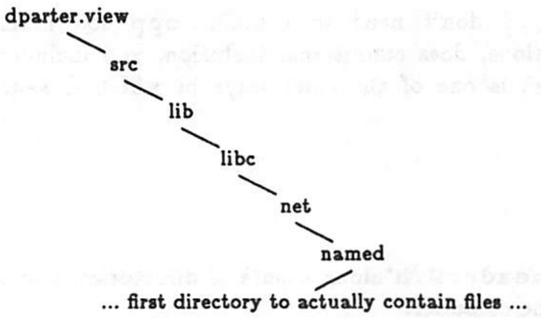


**Figure 4.** Sparse directory tree – stem of empty directories

A developer's *view* would reflect the contents of the source tree at the time that she started work on a task, plus whatever changes the developer had made. Thus, developers were isolated from each others' changes while actively working on the code; eliminating situations where "the code that I wrote last night worked, but somebody changed something that broke it this morning". Eventually developers would have to integrate their changes with the rest of the system, but being able to do it later, at their convenience, helped speed up their work. When major changes were integrated the call would go out to "update your views", for each developer to integrate the current system with the not-yet-integrated changes in her view.

This concept of a *view* also encompasses the snapshot of the system source at a baseline (an internal or external release that a code freeze is done for). The baseline snapshot is simply a view made at a particular point in time. Then, in conjunction with the space savings of hard-link sharing, it is natural to keep baseline views online for reference.

The RCS tree history object comprises the true master copy of the source; but we also kept a view of the latest checked in source at all times. In fact, this was the source that people would hard-link clone from. The checkin wrappers would first update the RCS tree, and then update the latest source tree views. This "current" view was convenient for browsing and reference; however, in retrospect I am not sure that it was worth the bother of maintaining in real time, particularly since, in an environment where there are multiple versions, it is an open question what the "current" version is. We maintained two different "current" versions, Development and Maintenance, for two different architectures.

**3.6 Object files:** Our rule is that the history object, the RCS and source trees, contains only source, not files that are generated from other files, such as object files or parsers generated by lex.

However, it is painful to remake large modules from scratch every time you make a view. Therefore, we have yet another set of parallel trees containing already generated object files for frequently used modules like the kernel. Developers making a view would have these object files placed into their tree. At baseline time all objects would be generated from scratch from the source; in fact, usually the objects were recompiled every night (finding many minor errors in doing so).

Like the current view, the object files would have to be updated at every checkin. This was awkward enough that, again, I am not sure that it was worth the effort.

## 4. BOXES:

Parallel trees provide isolation of the source code a developer is working on from other developer's changes. Boxes provide isolation of the environment a developer compiles her code in from the system she is running on.

Boxes are basically `chroot` environments for compiling source code in. I'll try to motivate the use of boxes by explaining some of the problems with one particular compilation tool, `cpp`, the C preprocessor, and show how boxes alleviate these problems.

**4.1 CPP include paths:** Hopefully I don't need to explain `cpp` too much: it is a preprocessor that expands macro definitions, does conditional inclusion, and includes other files into the body of C programs. The last is one of the chief ways in which C source files are embedded in a larger environment.

Cpp supports directives of the form

```
#include "header1.h"
#include <header2.h>
```

These look for files `header1.h` and `header2.h` along a *path* of directories, and include it in the input for subsequent processing by the compiler.

Typically, the default path is only one element long, containing the directory `/usr/include`. Other elements can be prepended to the path by using the `-I` directive, so a simple compilation command might look like

```
cc -I/cross/usr/include foo.c
```

defining an include path consisting of the directories

```
/cross/usr/include
/usr/include
```

The `"header1.h"` and the `<header2.h>` form differ only in that the current directory, `.`, is implicitly prepended to the include path for the first form.

With such a path, `cpp` would look first for `/cross/usr/include/header2.h`, and if that file did not exist, then `/usr/include/header2.h`. Thus, if you wanted to change or override some of the definitions in `/usr/include/header2.h`, you would simply have to create `/cross/usr/include/header2.h`, modify it, and ensure that all of your compilation commands use the appropriate `-I` options.

The gotcha here is that if any of your compilation commands do not have `-I/cross/usr/include` then some of your source will be compiled using the wrong header file. This can lead to annoying and subtle errors that can take a long time to find.

Part of the problem is that control of the options used for compilation is not centralized. However, the main problem is that the flavour of `cpp` we were using did not allow you to *eliminate* the default directories from the path. In our situation it was never appropriate to use any file from `/usr/include` on the host system.

Of course, it's easy enough to add a mechanism to eliminate the default path; but for various political, managerial, and compatibility reasons this feature was not allowed into our standard product. Moreover, `cpp` is only one of many programs that have this sort of path, and the change to allow elimination of the defaults would have had to be made to all of these programs.

**4.2 Chroot Boxes:** Boxes allow us to give each developer a copy of the entire environment she needs to work with; in this environment she can place executables and include files at the position in the file hierarchy they would occupy in the target system, and she is *guaranteed* that nothing from outside the box is accidentally picked up. This is easier than making changes to Makefiles spread all over the place, and the guarantee increases confidence in the product.

As stated above, "box" is just local terminology for a *chroot* environment. *Chroot* is a system call that allows the logical root of the filesystem to be moved to a subtree of the directory hierarchy, rendering files outside this subtree inaccessible. This prevents accidental access to outside files, and allows files to be placed at points within the file tree they wouldn't normally be allowed at.

Boxes were a rather natural idea for us: for a long time our site has used a box to encapsulate network traffic with the outside world, and UTX/32S, Gould's C2 certified operating system, expanded boxes into *restricted environments* [MSMYT86] as the basis for a certifiably secure system (although they are insufficient for higher levels of security). Conversely, boxes can be a security hole, used to spoof setuid programs into using incorrect data files. The security aspects of boxes are beyond the scope of this paper.

Boxes have been used elsewhere: Motorola SYSTEM V/68, for instance, uses a chroot box to compile the system.

We just carried boxes to the logical extreme: we provided a box for each developer (in fact, some developers used multiple boxes for different tasks). Each box contained as much of the environment as was required to build the system.

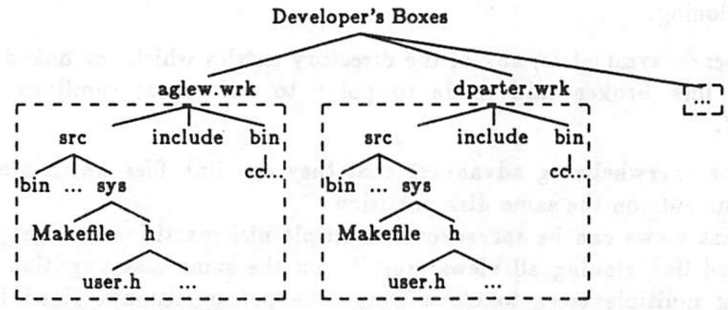


**Figure 5.** Boxes — parallel trees, including environment as well as source

Apart from the great amount of space they potentially consume, and apart from the security considerations, there are a number of other fine points to using boxes. Symbolic links are an annoyance. Our system typically has several symlinks:

```
/usr/include/sys -> /sys/h
/sys -> /usr.NP/sys
```

These are absolute paths, pointing to different places depending on whether you are inside or outside the box. Relative symlinks always work, but must be handled specially when moving directories around:

```
Moving a directory tree:
       IF symlink is internal to the tree being moved
       THEN
            move as a relative symlink
       ELSE
            move as an absolute symlink
       Convert back to preferred form.
```

However, with appropriate attention to a few little details like symlinks, boxes are a very agreeable development environment.

## 5. LINKS:

Parallel trees and separate boxes for each developer are convenient, but can easily consume many hundreds of megabytes of storage of disk space.

*Link* sharing of files made it possible to use boxes and parallel trees by dramatically reducing the space requirements. We owe the idea to [Yost85], but where Yost used links as his CM system, we used links only to save space in a CM system built around RCS. We used a descendant of Yost's `cpt` command to create link cloned trees; `cpio -l` may be used for the same effect.

"Link cloning" a directory tree means creating a new directory tree of the same structure where the ordinary files at the leaves are hard links to the files in the original tree. Files that are to be modified, in either tree, must have the link broken, and a new file created.

Greater security is provided if all of the files that are link-cloned are made non-writable. Writing into a link-cloned file, which is shared with other developers and the reference source, is the greatest of crimes!

Creating a developer's view consists of link cloning the prototype box environment, then link cloning appropriate parts of the current reference view of the source tree, both shared and private, and finally copying pre-generated object modules, into a new directory tree.

Originally, we used BSD symlinks to clone trees, but we moved back to Yost's original hard link cloning for several reasons:

1. Symlink cloning occupies an inode for each symlink, and thus occupies considerably more disk space than hard-link cloning.

2. Hard-link cloning is, in a sense, symmetric; any of the directory entries which are linked by hard links may have its link broken and made to point to new data; symlinks are unsymmetric. See Figure 7.

3. Finally, hard-links have the overwhelming advantage that they can link files which are in different chroot box environments on the same disk partition.

Symlinks have the advantage that views can be spread over multiple disk partitions; to get the maximum space saving from hard link cloning all views must be on the same disk partition (or you must spend effort updating multiple trees to clone from, one per partition). Hard link cloning saves disk space, but you need large disk partitions. Our system administrators did not like backing up >600MB partitions, particularly given our practice of running the latest kernels on the machines we worked on - leading to MTTFs slightly smaller than the time to do a level 0 dump on the development partition.

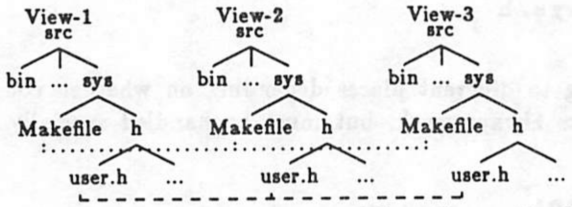


**Figure 6.** Parallel trees, with ordinary files hard linked

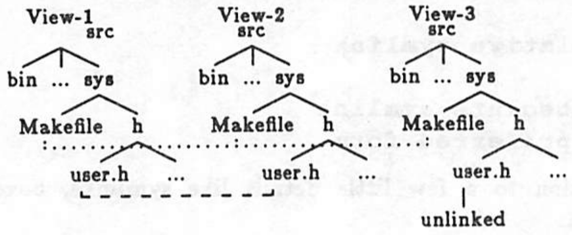


**Figure 7.** Parallel trees after a link is broken

There are some fine points to managing hard link cloned trees: As mentioned above, you have to make sure that link-cloned files are non-writable. This worked okay for source code. Making the binaries that we kept in the box environment non-writable created a few problems - chiefly in setuid install scripts that attempted to write *into*, instead of *in place of*, binaries in directories like `/bin`. Making object files, like pregenerated kernel `.o` files, nonwritable was a pain — many Makefiles and scripts broke in awkward ways — so the Btools gave up and made these "real", ie. copied, not cloned. These object files that were not shared between parallel trees were the biggest single space consumers.

Directories still take up a lot of space, even if all the files in them are hard link cloned. This can be alleviated by NFS mounting read-only directory trees within each box. In another attempt to share directories between boxes, I hacked in a variant of symlinks that could point outside of a chroot environment; this was easy to do, but never made it into the product because of security considerations.

Anything that attempts to reduce complexity by mounting directories must have operations convert between directories shared by mounting, and directories shared by linking.

## 6. OTHER FEATURES:

Although parallel trees, boxes, and link cloning are the main points of this paper, our configuration management and development system had other features. The importance of consistency checks, locks to prevent inconsistencies, and testing can never be underemphasized.

### 6.1 Consistency checks:

As in any redundant data structure, parallel trees need consistency checks. The CM team regularly ran scripts at night checking back and forth, ensuring that the "current view" corresponded to the latest checked in RCS source, had the appropriate permissions, etc.

### 6.2 Integration Testing:

Parallel trees and boxes give each developer an isolated environment in which she can work without interference from other developers' changes. This works fine up until the very end, at which point a developer's changes must be integrated with the rest of the system.

We try to avoid situations where several developers were integrating all at the same time by spreading integrations out over the schedule. Integrations are a serialization point.

**Figure 8.** Development and Integration Cycle

```
Check out files
Development cycle:
    Make modifications
    Test; repeat modify/test until acceptable
Optional Pre-Integration:
    Determine other people's changes
    Merge them into your source
    Test; repeat integrate/test until acceptable
Mandatory Exclusive Integration:
    Obtain exclusive lock on source
    Determine other people's changes
    Merge them into your source
    Test; repeat integrate/test until acceptable
    Check in
    Release lock
```

In addition to the testing done by the developer according to the above cycle, at times we found it useful to have an independent post-integration test cycle. In this approach, an integration tester would run a fairly well automated test suite on a new system; typically, three or four systems would be batched, the latest tested, and earlier versions tested only if the latest failed, to isolate the point of contamination. Finally, of course, there was the regular cycle of baseline and release testing.

Because of the importance and fragility of the kernel, the Ktools and KUtools strongly encouraged a sequence of tests to be run on newly integrated kernel modifications. This checkin testing, our integration cycle, and post-integration testing all reduce errors that might be introduced by the "splendid isolation" afforded by parallel trees and boxes for development.

**6.3 Locking:** Raw RCS and SCCS both provide locking for mutual exclusion on a file-by-file basis. Usually, however, a single change requires modifications to multiple files; this evolves naturally towards database transactions, and requires a locking protocol that covers multiple files.

Without locks, errors like this occur:

Figure 9. Errors without multifile locking
Programmer 1 has locked and modified 2 files, A and B,
whose changes depend on each other.
Programmer 1 checks file A in.
Programmer 2 makes a view, and gets the new version of A
and the old version of B.
Programmer 1 checks file B in.
The reference tree is consistent,
but Programmer 2's view is inconsistent.

We implemented a single lock for the entire source tree, held for the relatively short time it would take to make a view, acquire locks on a batch of files, or check in a batch of changes. The kernel itself was covered by a separate lock, since a kernel integration could take a long time. This lock would send mail to the next person on a waiting list when the lock was released.

At first glance you would think that a directory tree would lend itself well to a system of parallel, nested, locks. This isn't quite true, because a package of changes often includes files from widely separated parts of the tree. In any case, it turned out that locking of the trees was seldom a bottleneck.

**6.4 Prelogging:** RCS and SCCS let you write a note describing what you have changed when you check in a file. It is also very useful to write a note describing what you intend to do to a file when you check it out for editing. On several occasions I've had files checked out for editing for months, and completely forgot why I had checked them out.

We called this feature prelogging. The original Ktools had this. SCCS modification requests are a step in this direction.

## 7. MISCELLANEOUS OBSERVATIONS:

**7.1 Configuration Managers vs. Developers:** It's not a good thing to have happen, and it's probably not very good politically to admit that it happens, but a "Them vs. Us" mentality can evolve between the Configuration Management Team and Developers. I'm allowed to say this sort of fragmentation occurs because I've been both and am presently neither.

CM tools are usually written by people on the CM team to make their job easier. Trouble is, the CM tools have to be used by developers. Frequently, the things that the CM team found too hard to do are the things that the developers want. If the CM tools are hard to use, developers will avoid using them.

Example 1: name transformation - a developer does not want to have to type in a long name when checking in, or answer many questions about which file; she wants to check *this* file in her source into its corresponding RCS tree. But designing good default behaviour for name transformations is non-trivial, and too often the CM team will say that the general case is good enough. Worse, they will only provide one convenient idiom, and will not even permit long names to be used in full as a fallback if the developer does not like the idiom the CM team has provided.

Example 2: batch ability - programmers like programming. We like to be able to write a script that checks in everything — and, in fact, this can be a good thing, because such scripts are less prone to inconsistencies in the log messages. Some of our first tools (written by CMers) were strictly interactive; others made programming unnecessarily difficult, eg. by requiring the checker-inner to cd to the directory that contained the file, preventing the use of idioms like

`find . `*-predicate for a file I've changed* `-exec` *checkin* `{} ;`

**7.1.1 Networking:** Our site was networked using NFS, and CM work was spread across several machines. This was a lot better than earlier systems that required developers to copy their modifications to a central machine for CM activities; but more work needs to be done on distributed CM systems for sites too widely separated for NFS.

## 8. CONCLUSIONS WITH ABSTRACTIONS:

This paper has presented boxes, links, and parallel trees, mechanisms that can be implemented with the UNIX filesystem, which are a nice elements to wrap a CM system around RCS or SCCS with.

It helps to think of these things in the context of programming *development environments*, for which there are several basic operations:

1. *Separating* and *isolating* developers' compilation environments.
2. When moving code around you must *extract* the code, and everything it depends on, from its native environment.
3. Conversely, at the destination you must *integrate* the code with its new environment, hopefully eliminating redundancies.

Parallel trees and boxes provide a robust and automatic mechanism for *separating and isolating environments*. *Environment extraction*, however, we do in a rather in a rather ad-hoc fashion through the making of views. The script that made a view, for example, might know that if you were making a view of a test suite component, you would need the test suite libraries as well. I do not know of any CM systems that do a good job of environment extraction, although things like `make depend` and the `requires` cause of LISP are a beginning. Finally, *environment integration* we accomplish through our checkin procedures and is almost totally unautomated. It is much too easy to integrate by copying the old environment; programmers need to learn how to destroy things.

## 9. Acknowledgements:

Joan Eslinger and Jeff Chiu did the Ktools. Andy Glew did the Btools. Susan O'Neill, Angel Hernandez, and Dean Score did the Gtools. Todd Kinney and Leo Duggan did the Utools. Angel Hernandez did the KUtools. All of these people, as well as Dave Carlson, Jim Leo, Rob Anderson, and many others, influenced the evolution of our toolset, and served in the Configuration Management trenches.

## References

[MCBHJN86] D. Mackay, G. Ball, M. Crowe, M. Hughes, D. Jenkin, and C. Nicol, Paisley Institute of Technology, *A UNIX-based System for Software Configuration Management*, The Computer Journal, vol.29, no.6, 1986, pp. 527-530.

[MCD88] **System V/68 Release 3 Configuration Management Tools Software Release Guide**, Motorola Microcomputer Division Internal Document, November 1988.

[MosChri86] Dale Mosby and Steve Christiansen, Sequent Computer Systems, *Software Change Control System*, **Uniforum 1986 Conference Proceedings, pp. 184-197.**

[MSMYP86] Greta Miller, Steve Sutton, Mikel Matthews, Joanna Yip, and Tim Thomas, Gould CSD, *Integrity Mechanisms in a Secure UNIX, Gould UTX/32S*, **Proceedings of the Second Annual Conference of the American Institute of Aeronautics and Astronautics, 1986.**

[Yost85] David Yost, *A Rich Man's SCCS* or *The Cloned Tree Method of Revision Control* or *A Rich Person's Revision Control System* or *How I adapted the UNIX file system and tools that manipulate it to perform project revision control*, **USENIX Association, Summer Conference Proceedings, Portland 19855.**

# The DV System of Source File Management

*Peter Costantinidis, Jr.*

*Hamish Reid*

UniSoft Corp.
6121 Hollis Street
Emeryville, California 94608-2092
uunet!unisoft!{peterc,hamish}

*ABSTRACT*

DV is UniSoft's solution to the problem of managing large quantities of source, used by multiple projects and targeted towards a variety of systems and architectures. DV was designed to

- be simple, maintainable, and portable,
- eliminate source file duplication,
- support individual and insulated working environments, and
- aid release and version tracking.

Source files and DV system support files are maintained in a central *master tree*. A master tree is composed of *branch DV trees*. Each branch DV tree contains a set of related source files under SCCS. DV users access and modify source from within *derived DV trees* which are produced from one or more branch DV trees. A derived DV tree is said to *subscribe to* these branch trees. DV users add new files to their derived DV tree by adding the files to one of the subscribed to branch DV trees. The DV user invokes DV utilities to manipulate files. Where appropriate, some DV utilities will invoke SCCS utilities.

Associated tools provide the ability to enforce rigid separation between different project environments (e.g. prototype, porting, support, etc.). In addition, tools are available that can flexibly monitor and audit configuration and source changes at the site, branch, project individual, or file levels.

The prototype distributed version of DV (''Robacom'') supports the sharing of branches between remote sites over a variety of media (typically UUCP). Branches are *shadowed* remotely, with consistency necessarily achieved over time. This remote feature helps support multi-site project management, support, and product releases.

After more than a year of analysis of source file management problems, and many months of experimentation with a prototype implementation, an early version of DV became operational in late 1987. A stable and tested version of DV is now used by UniSoft for UniPlus$^+$ development, porting and support in Emeryville and London on a number of systems (including the MacII, Sun, Pyramid, Sequent, and Motorola Delta series).

## 1. Introduction

This paper describes DV, UniSoft's solution[1] to the problem of managing source files. DV is a suite

[1] DV is an internal tool, not an actual UniSoft product.

of software tools used to manage, control, and manipulate source (i.e. ASCII) files. UniSoft's primary use for DV is to manage UNIX[2] source and documentation files. These files are used by multiple porting projects targeted towards different system architectures.

This paper will describe DV's key features, how it is used, its system requirements, and how DV works. In addition, some present and proposed DV extensions are described.

## 2. DV Features

### 2.1. Centralization

All source files and DV system support files are maintained in a central master tree. Files used by the underlying source control system (SCCS) are located in the master tree. A user's working directory tree, called a *derived DV tree* , merely contains symbolic links to the master tree. Since a derived DV tree does not contain source files, system administration tasks such as backups are simplified. Only master trees need be backed-up.

### 2.2. Source file sharing

A goal of DV is to minimize the duplication of source files. Different products and projects may have some source files in common and others that are unique. DV allows different products and projects to share the source they have in common. This is accomplished by grouping related source files in the master tree under a single *branch DV tree* .

### 2.3. Branch DV trees

While DV does not require that related source files be grouped under a single branch tree, there are many advantages to doing so. With a well structured master tree, one could pick the minimal subset of branch DV trees when building a derived DV tree. One could restrict access to an entire branch DV tree, for instance, by closing the branch DV tree's top level directory. Responsibilities could be assigned on a per branch DV tree basis.

When a file exists in more than one of the subscribed to branch DV trees, the DV user specifies the desired order of precedence.

### 2.4. Insulated working environments (version locking)

As stated earlier, files in a derived DV tree are symbolic links to corresponding files in the master tree. By default, the symbolic links point to the most recently checked-in version of the corresponding file. The derived DV tree owner can specify that they are not interested in any new modifications made to a particular file or set of files. This is referred to as file version *locking* , or subscribing to a particular version of a file. In this manner, older version of a file can be accessed and used.

A derived DV tree owner can lock onto a particular version of a file. The DV user is then guaranteed that the locked file, within the derived DV tree, can not be modified, removed, or moved. The file in the derived DV tree can only change if the DV user unlocks it or locks it onto another version. Other DV users are unaffected by such a lock, they are free to manipulate the file in any way. Locking can aid in debugging by increasing the stability of the working environment.

A derived DV tree in which every file is locked is referred to as a "locked DV tree".

### 2.5. Reconstructing old releases

A locked derived tree's DVBOM data base can be archived in order to reconstruct the tree at a later date. By default, it will always be possible to reconstruct a derived DV tree, even if files to which it earlier subscribed have since been renamed or removed.

[2] UNIX is a trademark of Bell Laboratories.

## 2.6. DV Bill of Materials data base

Associated with each branch and derived DV tree is a DV *bill of materials* (DVBOM) data base. A DVBOM data base describes the content of the corresponding DV tree. There is a DVBOM entry for each file located in the DV tree. Each entry specifies a file's location in both the derived and the branch DV tree as well as the file's current version.

When a derived DV tree employs version locking on a particular file, the file's DVBOM entry is changed from a wildcard designation to that of a particular file version. The version fields in branch DV tree DVBOM data bases always refer to the version of the most recently checked in file.

A DVBOM data base can be used to take a snapshot of the current state of a derived DV tree. To do this, one would lock all files in the tree and then make a copy of the DVBOM data base.

Associated with each DVBOM data base is a *dvinfo* file. This file contains information such as:

- who owns the corresponding DV tree,
- what branches it subscribes to (for derived DV trees only),
- when it was created,
- where it is located, and
- why the DV tree was created.

## 2.7. File name mapping

The directory structure of a branch DV tree does not necessarily parallel that of a derived DV tree. Where differences exist, the branch DV tree's DVBOM data base specifies the mapping. This feature can be used to reorganize directories and as well to rename files.

File name mapping can be useful when handling third party source. One could arrange source in a branch tree according to the directory of structure used by the third party and arrange source in the derived tree according to the conventions used at the current site. One could compare new source releases from the third party directly against the source in the branch tree.

## 3. Using DV

The three basic things that a new DV user must know are how to:

- create a new derived DV tree,
- edit files, and
- add, move, or remove files.

These, and administration topics, are explained in the following sections. Some details have been omitted for brevity.

## 3.1. Creating

Creating a derived DV tree is a two step process. First, the user must build the DV tree's DVBOM data base. Then, the user constructs a directory tree according to the specifications in the DVBOM data base.

The DVBOM data base is built by the `dvmkbom` utility. Before running `dvmkbom`, one must first decide the branch DV trees to which the derived DV tree will subscribe. Usually, all derived DV trees associated with the same project will subscribe to the same branch DV trees. In such cases, the user can obtain branch subscription information from an existing derived DV tree's `dvinfo` file. The `dvnames` utility can also be used to produce the list of all branch DV trees.

In the following example, a DVBOM data base that subscribes to four branch DV trees is created:

```
$ dvmkbom -d usenix -r 'pwd' mtv M68020 m68k std5.3
usenix> This derived DV tree was created for the purpose of
usenix> demonstrating how to create a derived DV tree.
usenix> .
```

The string "usenix" is the derived DV tree's *DV name*. Every DV tree has a unique DV name. The DV name is used to identify the appropriate DVBOM data base. The "-r" option specifies the location of the derived DV tree.

The directory tree is constructed using the dvmkdvt utility. In the following example, the usenix DV tree's directory tree is created in the current directory:

```
$ dvmkdvt -d usenix -r `pwd`
```

The "-d" and "-r" are not necessary if the "DVNAME" and "ROOT" environment variables are defined. In addition, after the derived DV tree has been created, one can instead use the DV front-end utility, dv.

It may take a few minutes to create the directory tree...

## 3.2. Editing

Before a file can be edited, the user must check out the file using dvckout. After editing is complete, the file is checked in using dvckin. The dvinfo utility can be used to list all files checked out in the current directory or in the current derived DV tree.

Below is an example:

```
$ dv dvckout -Y test.c
dvckout: current file: "test.c"
1.1
new delta 1.2
1 lines
std5.3
dvckout(test.c)> This file was checked out in order to demonstrate
dvckout(test.c)> how to check out a file.
dvckout(test.c)> .
$ dv dvcdot test.c
This file was checked out in order to demonstrate
how to check out a file.
$ dv dvinfo
/usr/src/cmd/foo/test.c: 1.1 1.2 peterc 89/02/26 18:12:05
$ dv dvckin test.c
dvckin: current file: "test.c"
delta(test.c)> This file was added, checked out, and checked in in order to
delta(test.c)> demonstrate some DV commands.
delta(test.c)> .
1.2
0 inserted
0 deleted
1 unchanged
1.2
1 lines
```

When checking out a file, the name of the branch DV tree from which the file originates is printed. In this example, the file "test.c" comes from the "std5.3" branch DV tree.

When the "-Y" option is used, the user is prompted for a comment that will become associated with the checked out file. Such comments can be useful when another user wishes to know why a file has been checked out. The dvcdot can be used to print any comments associated with a particular file.

Besides displaying information about files being edited in the current working derived DV tree, dvinfo can also display editing information about any file subscribed to by the current tree that is being edited in any other tree.

When checking in a file, `dvckin` automatically prompts the user for a comment. Comments are terminated by an end-of-file condition or by typing a line containing only a period. There are also a limited number of *tilde escape* commands that can be invoked at the `dvckin` prompt.[3] These commands can be used for such things as invoking an editor on the comment or displaying the comment buffer.

## 3.3. Adding, moving, and removing

Files are added using `dvadd`, moved using `dvmv`, and removed using `dvrm`. DV insures that, when a file is moved or removed, derived trees locked onto the file will not be affected.

When adding a file, it is best to view the operation as adding a file to a branch DV tree.[4] The file is first copied to the desired location in the derived DV tree. Then, an appropriate branch tree is chosen and `dvadd` is invoked. In the following example, "`tesl.c`" is added to the "`std5.3`" branch DV tree:

```
$ dv dvadd -b std5.3 tesl.c
1.1
1 lines
$ ls -FC
tesl.c@         tesl.c.old
```

Using `dvmv`, files can be renamed, moved to other directories, or moved from one branch DV tree to another. In the following example, the file "`tesl.c`" is renamed to "`test.c`":

```
$ dv dvmv tesl.c test.c
$ ls -FC
tesl.c.old      test.c@
```

In the next example, the file "`test.c`" is removed:

```
$ dvrm -Y test.c
dvrm(test.c)> This file was created in order to demonstrate dvadd.
dvrm(test.c)> It is being removed in order to demonstrate dvrm.
dvrm(test.c)>.
```

These changes are immediately visible in the current working derived DV tree. Other derived DV trees will not observe these *configuration changes* until the `dvupdate` command is run within them.

## 3.4. Administration

DV provides administrative utilities for performing functions such as checking a DV tree for consistency, removing a DV tree, creating a master tree, creating a branch DV tree, and performing both regular and preventive maintenance on a master tree.

Regular maintenance on a master tree consists of locating and removing unused versions of files and maintenance of the branch tree DVBOM data bases. Preventive maintenance consists of reporting inactive DV trees, missing DVBOM data bases or DV info files, and also of checking branch DV trees for consistency.

## 4. How DV works

The master tree is a collection of branch DV trees. The master tree can span multiple directories with each directory containing one or more branch DV trees. The collection of directories comprising the master tree is called the master tree *search path*. The DV user specifies the master tree search path using the "DVMASTER" environment variable or by using the DV front-end utility, `dv`.

Branch DV trees contain the files necessary to support the underlying source control system (currently SCCS). In addition, a branch tree contains the targets of derived DV tree symbolic links. Files

[3] All DV utilities that prompt for user input support these tilde escape commands.

[4] Though not required, one would usually add the file to a branch DV tree subscribed to by the derived DV tree.

that are currently being edited are also located in the branch tree, the derived tree merely contains a symbolic link to the file. This makes it possible for multiple derived trees to subscribe to the editing version of a file.

Every master tree contains one branch DV tree named ''dv''. Branch DV tree DVBOM data bases are always maintained in this branch tree. Derived DV tree DVBOM data bases are maintained here by default.

A derived DV tree contains symbolic links to files located in subscribed to branch DV trees. These symbolic links may point to any version of a particular file. A mechanism exists such that the derived DV tree can always point to the last modified version of a file. It is possible to construct a derived DV tree containing copies of files rather than symbolic links.

Branch tree DVBOM data bases describe the location and latest version of all files in the branch tree. The `dvmkbom` utility builds derived tree DVBOM data bases by merging the specified branch tree DVBOM data bases into one data base. Where duplications exist, the order in which the branch tree names were specified dictates the order of precedence for discarding duplicates. In this way, many software configurations can be easily built.

Derived tree DVBOM data bases describe the location of all files within the derived tree, the branch tree from which files originate, where files are maintained within the branch tree, and the versions of files that are locked.

When a file is removed, by default its corresponding SCCS s-dot file remains. This enables derived DV trees to continue accessing the file. It is for this reason that derived DV trees are said to be disposable. Derived trees can be removed when no longer needed and regenerated later with a single command.

DV utilities manipulate DVBOM data bases, branch trees, and derived trees as appropriate.

## 5. DV requirements

DV requires a System V version of UNIX or UniPlus$^+$, or BSD 4.2 augmented by System V emulation libraries.

Symbolic links are a fundamental system requirement. There have been investigations into using DV without symbolic links. While it would not be impossible, there would be more system overhead (especially greater disk space consumption) and loss of functionality.

Using DV leads to an unusual number of small files (i.e. symbolic links). There is a tendency to run out of inodes long before running out of disk space. File systems should be built with more inodes than the default.

## 6. DV extensions

Because of the narrowness of DV's original scope, the simplicity of its basic foundation, and the flexibility of its implementation, DV has proven to be an excellent base for more general software management systems. This section describes some of the planned or proposed extensions to DV[5] and their use in a typical project environment.

### 6.1. Remote branch tree consistency management

We refer to this extension as *Robacom*. Robacom provides a system for managing remote branch trees and keeping them consistent. This system allows DV users at remote sites to share arbitrary sets of branch DV trees. Changes made to a branch tree at one site are propagated to all other interested sites.

Under the DV system of source file management, the smallest unit of files that one might ''subscribe to'' is a *branch tree*. If one subscribes to any files in a branch tree, one must subscribe to all files in that branch tree. A problem exists when users at multiple sites wish to subscribe to the same branch trees (i.e. source files), in effect, sharing the same branch trees. The goal is to solve the problem of remotely sharing one branch tree and then repeatedly applying this solution to all branch trees that need to be shared.

[5] Mention of an extension here in no way guarantees that the extension will ever be developed.

The solution is to duplicate a branch tree at all sites wishing its subscription. One site is designated the primary site for a particular branch tree. The primary site for a given branch tree is responsible for tracking and controlling changes for that particular branch tree. Sites with duplicates of this branch tree are designated secondary sites. A site can be a primary site for one branch tree while being a secondary site for another.

A prototype of Robacom (called *Roba-hack*) was developed by the authors in October, 1988. Using this prototype we are able to "automatically" keep sources between our Emeryville and London offices consistent.

### 6.2. Software Configuration Management

We refer to this extension as *SCM tools*.[6] SCM is a formalization of the use of DV in a project environment. SCM defines a set of techniques on the use of DV as well as a set of tools to implement those techniques. The SCM tools are designed to augment the standard DV tools. They help with such things as deriving new DVBOM data bases, tracking software changes, and coordinating related derived DV trees.

A prototype version of the SCM tools was developed in the summer of 1988. Using this prototype, changes in the master tree are more easily managed and derived DV trees related to the same project are assured of sure subscribing to the same versions of files.

### 6.3. DV without symbolic links

With this extension, two different versions of DV would be supported. The two versions are referred to as the *optional* version and the *nosymlinks* version. The *optional* version would fully support a DV tree containing copies of files instead of symbolic links. In addition, DV trees containing symbolic links would also be supported. This version could only be supported on systems that support symbolic links. The concept of a DV tree *type* would be introduced. The DV tree owner would somehow specify that the DV tree is to be of type *copy* or of type *symlink*.

The *nosymlinks* version of DV is one completely without symbolic links. Altering of the internal master tree structure would be required. This version of DV would be portable to UNIX systems that do not support symbolic links.

This extension exists only on paper.

### 6.4. Tracking related files

Sometimes there are intersections between different branch trees. These intersections are referred to as *siblings*. When dealing with multi-targeted UNIX source, there is a fairly consistent set of system specific source files. This is especially true when files are cloned (using `dvclone`).

When manipulating a file in certain ways it can be useful to have automatic notification of all the file's siblings. This problem can be generalized such that one can create arbitrary file relationships. Thus, if one had an arbitrary set of closely related files, modifying one would generate a message informing the user of all the other related files.

A mechanism for tracking siblings has been proposed, though no prototypes yet exist.

### 6.5. Supporting non-ASCII files

In those situations where not all source is available, where one must build a product using a mix of source and object files, DV should be able to maintain both types of files. Given DV's master tree structure, this situation is not all that difficult to handle. To implement this extension would not require adding functionality to DV as much as it would require the suppression of some of the current actions.

[6] Pronounced "scumtools".

## 6.6. Graphical User Interface

DV could benefit greatly if it had a graphical user interface. Not only would it look sharper, the learning curve of new and novice users might be reduced. For instance, locking onto a specific version of a file would be more convenient if one could select the file with a mouse, display a menu listing the available versions, and then select the desired version for locking. Using `dvprs` to list file versions and then `dvsetver` (with the appropriate command line options) to lock onto a version requires more sophistication on the part of the user.

## 7. DV development history

| | |
|---|---|
| 08/13/86 | Engineering brainstorming session (discussion of symbolic link idea) |
| 10/03/86 | Proto-type system developed |
| 10/07/86 | Working-group brainstorming session (first identification of tools and preliminary master tree structure) |
| 05/18/87 | Design phase begins full-force |
| 09/11/87 | Work begins on tools to support a single derived DV tree |
| 03/31/88 | First version of DV placed in use; many maintenance and enhancement releases to follow |
| 10/27/88 | Remote branch tree (Robacom) preliminary design complete |

**Acknowledgments**

DV would not be so successful if it were not for the many patient users and their willingness to make suggestions. Our thanks are also due to Jak Mang and Teresa Costantinidis for their invaluable assistance in reviewing this paper.

## A. DV command list summary

This section lists various DV utilities by category:

### A.1. Source file management tools

These tools are used to regulate access to source files:

`dvadd` – add files to a branch DV tree
`dvckin` – check in file changes
`dvckout` – check out a file for editing
`dvclone` – clone files into another branch tree
`dvget` – get a version of a file
`dvinfo` – print information about a DV tree
`dvmv` – move or rename files
`dvprs` – print an SCCS file
`dvrm` – remove files
`dvunrm` – undo the effects of a dvrm
`dvsetver` – set or change versions of subscribed-to files
`dvunget` – undo the effects of a dvckout

### A.2. DV tree tools

These tools are used to create, manage, and remove derived DV trees:

`dvmkbom` – derive a DV tree DVBOM data base and create a DV info file
`dvmkdvt` – construct a derived DV tree
`dvrmtree` – remove a derived DV tree and associated files
`dvupdate` – reflect recent master tree changes or change branch subscriptions

### A.3. Administrative tools

These tools are used to administer DV trees in general:

`dvbsort` – clean up a DVBOM data base
`dvck` – DV tree consistency check and repair
`dvlck` – manage DV exclusive-lock files
`dvmkbt` – create a branch DV tree
`dvvfile` – list unused versions of files

### A.4. Miscellaneous tools

The following tools do not conveniently fit into the previous categories:

`dv` – front end for the DV subsystem
`dvbdiff` – show differences between DVBOM data bases
`dvnames` – list active DV names
`dvother` – list other branches containing file with same name
`dvsub` – print subscription information

## B. Sample DV info file

Below is the DV info file that was created by one of the earlier examples:

```
BRANCHES=mtv:M68020:m68k:std5.3
CREATED=2408A5C7
DVMASTER=/v3/MASTER
DVNAME=usenix
MAIL=peterc
OWNER=peterc
ROOT=/usr/ra/tools/peterc/dvtree
VERSION=DV:dvmkbom.sh 3.29
%%
This derived DV tree was created for the purpose of
demonstrating how to create a derived DV tree.
```

# Controlling Software for Multiple Projects

*Dale Miller*
Sterling Software Intelligence Military Division
1404 Fort Crook Road South
Bellevue, NE 68005-2969
UUCP: *dale@ssbell*
INTERNET: *dale%ssbell.uucp@uunet.uu.net*

**ABSTRACT**

Configuration Management is a management procedure including the components of configuration identification, control, and status. This paper presents the system that is being used for several defense contracts at Sterling Software Intelligence and Military Division (IMD) for projects using a *UNIX*[1] operating system. Features of the system include:

- Identifying and controlling software for more than one project,
- Maintaining baselines for multiple projects that are always available,
- Guaranteeing that project baselines are not altered by team members,
- Permitting only authorized team members to have access to their project baseline and development area,
- Identifying all the software that is being developed,
- Identifying what items are missing from each project baseline, and
- Reporting the current status of a project.

## 1. Introduction

The system uses the *Source Code Control System (SCCS)* with a set of front end data files and programs. The system identifies, controls, and accounts for the software and data files for several defense contracts.

## 2. Motivation

*SCCS* is a set of programs which allow developers to track the modification of text files. The system provides the ability to retrieve any version of a source program and allows the developer to produce new versions of programs while supporting the older versions. The key features of the *SCCS* system are:

| | |
|---|---|
| *Storage:* | The use of disk storage is minimized by only recording the modifications made to the systems rather than maintaining separate copies of each version of a file. |

[1]UNIX is a registered trademark of AT&T.

*Protection:* Access to files is controlled by *SCCS* and modifications to particular files (and particular versions of those files) may be restricted to certain individuals.

*Identification:* Each file can be stamped with information identifying the author, version and the date and time of modification. This information can be propagated to the object files produced from these files.

*Documentation:* The system prompts the user for a reason for all modifications and records who made the change, what was changed and when it was changed. This information can later be retrieved and used to report modifications of programs [Bazel85].

*SCCS* does not adequately provide for the management and control of large configurations, nor for cooperative development among several programmers [Black89]. Tools were needed to expand the control and usefulness of *SCCS* for multiple projects of various sizes.

## 3. Overview

Sterling Software IMD has several contracts that require deliveries on workstations using UNIX System V. Sterling has an ethernet LAN with several workstations and a VAX[2] 750. The VAX 750 is the host on the LAN used for the configuration management. All work that is done is controlled on the VAX 750. The remote access commands of **rcp** (remote copy) and **rsh** (remote shell) are configured on all UNIX machines on the LAN. Root access is restricted to the system administrator and the master SCCS administrator on the VAX 750.

## 4. Security Concerns

The configuration manager assigned to a project must be responsible for their project baseline. It must be impossible for team members of a project to alter a baseline. People that are not part of a development team should not have access to a baseline. An audit trail should be available to show the activity on a baseline.

## 5. Design

The design consist of several data files and programs. The *UNIX Programmer's Manual, Volume 5, Languages and Support Tools* [AT&T5], suggests using an **SCCS interface program** to permit UNIX system users to use SCCS commands upon the same files. An SCCS interface program was specifically designed for the system at Sterling IMD.

The interface temporarily grants the necessary file access permissions to project members. One user must be chosen as the "owner" of the SCCS files for all projects and be the "**master SCCS administrator**." The master SCCS administrator is "dale" at Sterling IMD. Each project has a **project SCCS administrator**.

[2]VAX is a trademark of Digital Equipment Corporation.

Since other users of SCCS do not have the same privileges and permissions as the master administrator, the other users are not able to execute directly those commands that require write permission in the directories containing the SCCS files. Therefore, the projects-dependent program is required to provide the interface.

The interface program

- must be owned by the master SCCS administrator,
- must be executable by the new owner, and
- must have the "set user on execution" bit "on" [see **chmod**(1) in the *UNIX Programmer's Manual-Volume 1: Commands and Utilities*].

Then when executed, the effective user ID is the user ID of the master administrator.

The interface is a program that intercepts SCCS calls before executing the real SCCS program. The interface program is linked to the SCCS commands. In this system the interface program uses several data files to determine if the file to be controlled by SCCS is a part of a project and if the user is a project team member or the project SCCS administrator. If the file is part of the project, the program checks the UID of the user to determine if the user is the project SCCS administrator.

Only a project SCCS administrator has permission to do the following SCCS commands: **admin**, **cdc**, **comb**, **delta**, **rmdel**, **val**, and **vc** (see [AT&T1] for explanations of these commands). All other project members are restricted from doing these SCCS commands.

Links are made to the interface program for the following SCCS commands: **admin**, **cdc**, **comb**, **delta**, **get**, **prs**, **rmdel**, **sact**, **unget**, **val**, and **vc** (see [AT&T1]). All the links are in /usr/lbin with the file permission modes of -rwsr-xr-x (4755 -- read, write, set user on execution for owner; read and execute by group; read and execute by others). The real SCCS commands are in /usr/bin with the permission modes of -rwxr-x--- (750 -- read, write, execute by owner; read and execute for group; no access for others). The SCCS commands in both the /usr/lbin and /usr/bin locations are owned by "dale" with the group "CM." All users of the VAX 750 have /usr/lbin in their PATH definition and both "root" and "dale" have /usr/lbin in their path before the /usr/bin entry. This prevents "root" from having SCCS access to the baseline unless root is identified as a team member. In this environment the interface program invokes the desired SCCS command and causes it to inherit the privileges of the master SCCS administrator for the duration of that command's execution.

If the file is not part of the project all SCCS commands are available to the user. This permits a person to use SCCS on their own files if desired.

A PROJECT variable is set and exported to the users environment for each team member in their **.profile** file. This variable must be set to identify the project for the person. If a person is a member of several projects they must set the value appropriately.

The interface program uses several data files. The first file is the **masterfile**. The masterfile identifies all the projects being controlled. If a person is identified as a member of a project in the masterfile, that person has permission to access (by a restricted set of SCCS commands) the baseline kept for the project. The access rights to the baseline of a project are the same except for the project SCCS administrator.

In this design the order of the data lines in the masterfile is important. The following is an example masterfile with one project defined:

```
# %W% %G% - CM - Sterling IMD
#
# This is a sample masterfile with one project.
#
# FORMAT:
#  $$project id
#    project administrator user id
#    project configuration management group id
#    project quality assurance user id
#    project project manager user id
#    project CM root directory
#    project pathfile
#    project SCCS logfile
#    valid user list
################################################
#
#  VAS - Value Added Software
#
################################################
$$VAS
103                         # SCCS administrator - Dale Miller
90                          # CM Group id
114                         # QA - Lisa Hill
404                         # Project Manager - Dave Krejci
/cm/vas                     # Project CM root directory
/cm/vas/adm/.pathfile       # VAS Pathfile
/cm/vas/adm/.sccslog        # VAS CM SCCS Logfile
#
#Valid project members
#
101                         # Kent Landfield
102                         # Chris Olson
107                         # Kevin Brainard
405                         # Roger Riley
################################################
```

The second data file is the **pathfile**. The complete path name to the pathfile for a project is identified in the masterfile. The pathfile identifies the complete path name to each **projectfile** for a project. The pathfile exist because a project may have more than one projectfile, and the project SCCS administrator should have direct access to naming and controlling each projectfile. In this way the master SCCS administrator (dale) may identify various projects in the masterfile with a level of indirection to the projectfile(s) that the project SCCS administrator controls. The master SCCS administrator only has to identify each project in the masterfile and set up the initial files identified. Then the project SCCS administrator has control of their project by using the system. All project SCCS administrators are in the CM group, thus allowing different projects to be assigned to different Configuration Management staff members.

A projectfile identifies files being controlled for a project. Each projectfile is maintained by the project SCCS administrator. The projectfile contains the filename, path, subsystem, and executable.

The filename is limited to 14 characters (due to a limitation of Unix System V) including the "s." for SCCS files. All filename entries should be unique (in the current design, the SCCS interface program uses the first match that it finds in this file). If a file is not controlled using SCCS, the filename should not begin with the "s." prefix.

The path is the path excluding the project CM root directory that is defined in the masterfile. The path for each filename does not begin with a slash (/) because the programs prefixes it with the project CM root directory path found in the masterfile.

The subsystem and executable fields contain values that helps in the selection of files for information retrieval in various reports.

The following is a sample from a projectfile:

```
# "%W% %G% VAS Software Configuration Index"
# This file contains sample data lines for the VAS Project.
s.bell.c            lib/vnas/s.bell.c             VNAS       libvnas.a
s.clr_frm_a.c       lib/vnas/s.clr_frm_a.c        VNAS       libvnas.a
s.clr_scrl_a.c      lib/vnas/s.clr_scrl_a.c       VNAS       libvans.a
s.convert.c         lib/vnas/s.convert.c          VNAS       libvnas.a
s.hide_mouse.c      lib/vnas/s.hide_mouse.c       VNAS       libvnas.a
s.int_window.c      lib/vnas/s.int_window.c       VNAS       libvnas.a
s.vnas.mk           lib/vnas/s.vnas.mk            VNAS       makefile
s.add_hosts.c       src/teladm/s.add_hosts.c      TELADM     teladm
s.alloc_fld.c       src/teladm/s.alloc_fld.c      TELADM     teladm
s.teladm.h          include/s.teladm.h            TELADM     header
s.teladm.mk         src/teladm/s.teladm.mk        TELADM     makefile
### End of sample data from VAS project file
```

This information permits the SCCS commands to operate for the project using the Configuration Management directories.
Using the information in the above sample files, Kent Landfield, user 101 could do the following:

```
PROJECT=VAS
get  -e  s.teladm.h
```

and the interface program would determine that Kent is a valid team member of the VAS project. The interface program then determines that the baseline root directory for the VAS project is /cm/vas from the masterfile and then checks the projectfile for the filename of s.teladm.h. It finds the data record and determines that the file to get is located at /cm/vas/include/s.teladm.h and retrieves the latest version for edit if it is not already out for edit.

By having the interface program linked to the SCCS commands and having all the files identified for a project in the data files it is never necessary for the development team or the project SCCS administrator to ever enter the complete path name to a file. This saves on key strokes and assures that the proper file locations are used.

It is important to note the following file permission modes for this example:

```
    drwxr-x---  dale    CM      /cm
    drwxr-x---  dale    CM      /cm/vas
    -rw-r--r--  dale    CM      the masterfile
    -rw-------  dale    CM      VAS projectfile
```

Besides the above data files, a sccslog contains data for important SCCS commands attempted or completed for a project. The interface program determines the name of the sccslog from the masterfile. The sccslog should have read access by the configuration management group and must have write access by the master SCCS administrator (dale). The information that is stored in the sccslog is the UID of the user, the date and time, the SCCS command and its parameters. The interface program logs a subset of the SCCS commands. Commands that are used for informational purposes only (i.e. prs and sact) are not logged.

## 6. Experience

Originally the design was to handle only one project. To handle one project the interface program was hard coded to look at one projectfile. However, several UNIX based projects forced the need to be able to handle multiple projects. This was permitted by enhancing the interface program to make use of the PROJECT variable and the masterfile. The original design was ported and is still being used in Germany to handle a project with over 1200 source files on a workstation.

Using the interface program to SCCS with the data files is helpful, but more helpful features are also readily available.

Various tools have been written to complement the project SCCS files. Several of these tools are used to validate the data files. The masterfile has a program named **ck_master** that both verifies and displays the contents of the masterfile. Depending on the user of ck_master, different information is available. One option (-l) returns information only to the project SCCS administrator or the master SCCS administrator. This information includes the project base directory, pathfile, and sccslog file names. The team members who are not SCCS administrators may use ck_master to determine what projects they have access to and who the other team members are.

Several **awk** (see [AT&T5]) programs have been written to verify the content of the projectfiles.

A projectfile is the file that identifies all files that are part of the baseline for a project. By keeping the projectfile current it permits status accounting to be done on the baseline.

Several programs make use of the projectfile data. Programs that access information from the project baseline directories must have the "set user on execution" bit "on" so that when executed, the effective user ID is the user ID of the master SCCS administrator. An example is "**check_path**" which is an aid to the programmer and management in extracting information from a projectfile. Several options are available which permits record selection by filename, executable, subsystem, and change control status. For example, using a "-c -e header" option gives a list of all files identified as a header file with change control status. The change control status reflects either the file permission modes or the fact that the file has not been placed under change control yet.

At Sterling a VAX 750 is used for controlling the baseline for various projects and for development. The development directories are *mirror* copies of the CM directories. For the VAS project, the CM directories are relative to /cm/vas and the development directories are relative to /usr2/vas.

Several procedures and tools have been written to aid the mirror development of the baseline for a project. On the VAX 750, only the project SCCS administrator can alter the project baseline. The team members may use the SCCS get command, but this does not regulate which directory the team member does the get in. The shell script "**check_out**" was written to maintain a "mirror" of the baseline for development. In the example earlier where Kent Landfield did a "**get -e s.teladm.h**" the results would have been a teladm.h file for edit in his present working directory. By using "**check_out s.teladm.h**" a file is checked out with Kent as the owner in the /usr2/vas/include directory. It is the responsibility of the project SCCS administrator to set up the CM baseline and the "mirror" development directories as they are identified.

To complement the "**check_out**" program, there is a "**check_in**" program that only project SCCS administrators have access to. If the file is not in the current directory, check_in determines the development location for the file and changes the directory to that location before doing an SCCS **admin** or **delta** on the file. Check_in also calculates and displays a check sum value that is used to verify the file. Check_in also alters the name in the (p.) file that SCCS uses to determine if a file is out for edit, to the name of the project SCCS administrator so that a delta may be done. Finally, check_in does a get in the development location to validate that the **admin** or **delta** was successful and to have a read only copy of the file available for the team's reference and recompilation.

The VAX 750 is not the machine used for testing of the baseline. The testing of the software is done on various workstations on the LAN. The VAX 750 permits a central location for maintaining and accessing the baseline.

To place a complete copy of a project baseline on a workstation connected to the LAN several tools have been developed. These tools make use of the masterfile and projectfile data and permit accurate transferral of the selected versions of the baseline files. It is important to note that the SCCS files (s.) are kept only on the VAX 750.

## 7. Generation of Reports

Status accounting for a project is controlled by the contents of the projectfile. By having a file that identifies the elements of a baseline it is merely a matter of writing tools that use the data in the files and the baseline files to produce information. More data elements may also be added to the projectfile as desired.

The following elements are included and used in several reports at Sterling Software IMD:

- The Computer Software Component Identifier (CSCI) number and name,
- Names of each unit within a source file,
- Total lines of Preliminary Design Language (PDL),
- PDL lines not including comment lines,
- Total lines of Code,
- Code lines not including comment lines,
- Time used for the inspection process,
- Date unit and source files were inspected,
- Date unit and source files were unit tested, and
- Flag showing if a file is in the baseline.

The following is the sample VAS projectfile including the above data elements:

```
# "%W% %G% VAS Software Configuration Index"
# This file contains sample data for the VAS Project.
#
# In order to use the facilities provided, you must
# enter the necessary information into this file.
#
# -----------------------------------------------
# There are 6 types of lines in this file.
#   1. Comment line              (# )
#   2. CSCI Number/Name line     (###)
#   3. File Name line            (#-)
#   4. s.filename line           (s.)
#   5. Unit Name line            (#+)
#   6. Line to halt processing   (*)
#
# For more information read the man page on "projectfile"
# The following is sample data:
# -----------------------------------------------
### LLCSC 1.1.1 VNAS User Interface
#-bell 34 6 75 23 25 09/22/88 10/27/88 yes
s.bell.c          lib/vnas/s.bell.c          VNAS      libvnas.a
#-clear_form_area 46 13 108 37 25 09/22/88 10/25/88 yes
s.clr_frm_a.c     lib/vnas/s.clr_frm_a.c     VNAS      libvnas.a
#-clear_scroll_area 0 0 113 50 15 11/15/88 -------- yes
```

```
s.clr_scrl_a.c    lib/vnas/s.clr_scrl_a.c  VNAS      libvans.a
#-convert 0 0 197 139 45 09/22/88 10/26/88 yes
s.convert.c       lib/vnas/s.convert.c     VNAS      libvnas.a
s.hide_mouse.c    lib/vnas/s.hide_mouse.c  VNAS      libvnas.a
#+hide_mouse 0 0 0 0 15 09/22/88 10/27/88 yes
#+show_mouse 0 0 0 0 10 09/22/88 10/27/88 yes
#-initialize_window 145 76 547 394 50 09/22/88 11/07/88 yes
s.int_window.c    lib/vnas/s.int_window.c  VNAS      libvnas.a
#-makefile
s.vnas.mk         lib/vnas/s.vnas.mk       VNAS      makefile
### TLCSC 1.3 User TELNET Administration Software (UTAS)
#-add_hosts 299 217 338 181 25 11/17/88 -------- yes
s.add_hosts.c     src/teladm/s.add_hosts.c TELADM    teladm
#-alloc_field 0 0 71 12 25 10/19/88 -------- yes
s.alloc_fld.c     src/teladm/s.alloc_fld.c TELADM    teladm
#-header 0 0 176 131 20 10/19/88 -------- yes
s.teladm.h        include/s.teladm.h       TELADM    header
#-makefile 0 0 556 495 45 11/21/88 -------- yes
s.teladm.mk       src/teladm/s.teladm.mk   TELADM    makefile
### End of expanded sample VAS project file
*
```

The above data file includes a lot of information that can generate many statistics; average size of source files, ratios of PDL lines to code, total code counts, number of files inspected, tested, under change control, percentages of work done, and other things.

From experience with a large project, it takes a large amount of effort to maintain a large projectfile with accurate information. Collecting many data items may give excellent statistics but it is more worthwhile to write good tools to generate current and accurate statistics from the baseline then to insert data into the projectfile that may be incorrect after the next delta.

One prime example of a tool to get information from a project baseline is **getbase**. This shell script makes use of the SCCS interface programs and the data files and a command regular expression that denotes the operation desired on the baseline. Getbase allows the user to execute a command across the entire baseline for a project.

Its usefulness is shown here by examples:

**getbase "egrep 'ifdef|ifndef' | grep # | sort | uniq"**

The above getbase command would produce a unique list of all the ifdef and ifndef lines used in all the baselined files for a project.

**getbase "clist -h" -emakefile**

This example would generate a clisting (clist is a print program that is used by Sterling IMD including line numbers, line wrapping, and other features) of all the makefiles identified in the projectfile with a header line containing the file name.

Other tools are available to generate code counts, labels for Software Development Folders, detailed listings of what files are out for edit, and listings of what files are in the baseline including version number, dates, and path names.

## 8. Documentation

The commands, files, and application programs are documented in **man** (manual) pages on the VAX 750. Certain distinctions of purpose are made in the headings:

| | |
|---|---|
| (1) | Commands of general utility. |
| (1M) | Commands restricted to SCCS administrators. |
| (4) | File formats. |
| (8) | Project system maintenance programs. |

The following are included in the man pages:

| | |
|---|---|
| **check_out(1)** | Get for edit a version of a project SCCS file. |
| **check_path(1)** | Search and print information from projectfiles. |
| **check_qa(1)** | Search and print detailed information from projectfiles. |
| **ck_master(1)** | Verify and display masterfile contents. |
| **clean_files(1)** | Optionally detab, sscb, and clist a file. |
| **cm_intro(1)** | Introduction to CM commands, files, and programs. |
| **cm_steps(1)** | Overview of CM procedures used at Sterling IMD. |
| **comments(1)** | Extract comment and non-blank lines from C source and generate counts of code, comments, etc. |
| **counts.awk(1)** | Generate source statistics using comments(1) as input. |
| **detab(1)** | Remove tabs from a file. |
| **diffit(1)** | Diff item against SCCS files. |
| **getbase(1)** | Get information from project baseline. |
| **getctags(1)** | Get function information from project baseline. |
| **labels(1)** | Generate labels for project Software Development Folders. |
| **sscb(1)** | Sterling Software C program beautifier. |
| **bkup(1M)** | Tape archive backup program for /cm directories (uses voltar). |
| **change_out(1M)** | Change the logname of user of impending delta. |

| | |
|---|---|
| **check_in(1M)** | Admin or delta a version of a project SCCS file. |
| **ck_data.awk(1M)** | Check data in a projectfile. |
| **prt_Docs.awk(1M)** | Format contents of Docs file for printing. |
| **reorg_path(1M)** | Beautify project projectfile. |
| **transfer_cm(1M)** | Get project baseline to transfer. |
| **whats_in(1M)** | Report what files are under CM control. |
| **whats_out(1M)** | Report what files are checked out for edit. |
| **Docs(4)** | Data file for document revision records. |
| **masterfile(4)** | Configuration project master file. |
| **pathfile(4)** | Configuration projectfile(s) pointer file. |
| **projectfile(4)** | Configuration project file. |

## 9. Future Plans

No system is perfect. The system at Sterling IMD is not, but it has proven to be successful and continues to be enhanced. Several items that are on the list for the future include:

- Adding more tools,
- Use of distributed /cm directories on a LAN,
- Making the package more portable to other users, and
- Releasing the software and documentation as public domain.

## 10. Conclusions

SCCS is a good tool for version control and insuring that only one person is working on a file at a time. However, it is limited in its success to control a baseline for a product to be delivered. By creating an interface program and data files the usefulness of SCCS can be greatly enhanced. The design is not complex and permits more general tools to be created in the future for even greater benefits.

## 11. Acknowledgments

The author wishes to thank Kent Landfield who wrote the interface program to SCCS and a collection of library routines to access the different data files. Kent is a major reason this paper can be presented.

The author also thanks John Stannard, the Director of Configuration Management/Quality Assurance for Sterling Software IMD, for his support in this presentation.

## 12. References

[Bazel85] Rudy Bazelmans, *"Evolution of Configuration Management," ACM SIGSOFT Software Engineering Notes*, vol 10, no 5, pp. 37-46, October 1985.

[Black89] Eric Black, *"Software Configuration Management with an Object-Oriented Database"*, Proceedings of the Winter 1989 USENIX Conference, USENIX Association.

[AT&T1] AT&T, Volume 1, *Commands and Utilities, UNIX Programmer's Manual*, CBS College Publishing's Unix System Library, 1986.

[AT&T5] AT&T, Volume 5, *Languages and Support Tools, UNIX Programmer's Manual*, CBS College Publishing's Unix System Library, 1986.

# Transcending Administrative Domains by Automating System Management Tasks in a Large Heterogeneous Environment

*Melvin J. Pleasant Jr.* (pleasant@rutgers.edu)
*Center for Computer and Information Services*
*Laboratory for Computer Science Research*
*H019 Hill Center - Busch Campus*
*Rutgers University*
*Piscataway, New Jersey 08855-0879*
*(201) 932-2023*

*Eliot Lear* (lear@net.bio.net)
*IntelliGenetics, Inc.*
*700 East El Camino Real*
*Mountain View, California 94040*
*(415) 962-7300*

## ABSTRACT

One could imagine a future in which there exists but one computer with infinite resources. Until such a time there will be distributed systems and the need for software distribution systems. This paper reflects over two years of experimentation and development that has occurred at Rutgers University involving various software distribution schemes. The goal of our first attempt was to free system administrators for more interesting projects by eliminating as many menial tasks as possible. The second incarnation reflects our desires to provide a service to entities outside of our administrative domain. Both versions of our system use *Track* as the underlying mechanism to distribute software and automate system administrative procedures.

## 1 Introduction

**The idea of distributed systems was first conceived when it became clear that mainframes could never be big enough for the tasks at hand, and that it would be relatively inexpensive to put a large amount of power on an individual's desk. Unfortunately, a problem not addressed by the mad rush of vendors into the workstation arena was that of software maintenance and administration. Looking back at our own mainframe experiences, we had problems keeping software up to date and synchronized on only three DECSYSTEM-2060 systems, each with its own dedicated system administrator. Still, we could only partially see the large clouds looming before us.**

These problems became more visible as we began our move to UNIX†[1] based workstations. A small modification in a program would immediately translate itself into a large investment of time and manpower copying the modified program from machine to machine. A system being down at the wrong moment often meant that it would not receive an updated configuration file. Bringing a new workstation online often involved scouring all others for the most recent version of some file. Worst of all, troubleshooting software bugs became extremely difficult due to continually arising inconsistencies between sources and installed executables.

It soon became clear that a mechanism to distribute software and automate simple administrative procedures was needed. We began by reviewing current procedures in conjunction with the various types of software maintenance utilities available in UNIX. We found these utilities to be lacking in that they worked well only for a small number of files involving only systems of like architecture, all configured alike. Clearly we needed a more flexible mechanism that would have and handle the following properties and tasks:

- A master system, known as the file librarian, that knows as little about its clients as possible. The addition of new clients should require little, if any, change on the master system.
- An update mechanism that must deal with temporarily downed clients. Such an error condition should not affect any other part of the distribution system.
- A distribution system that requires minimal maintenance. Our goal is not to laterally shift the usage of our human resources onto the task of maintaining the distribution system, but rather to significantly reduce the overall use of human resources for menial tasks. Under normal conditions, it should not be necessary for system administrators to modify the distribution system itself.
- A distribution system which must have the ability to fully update and configure a newly installed machine that has been minimally configured manually.
- A distribution system which must have the ability to service multiple architectures.

These were only an initial set of goals, as we had very little experience dealing with distributed systems on a large scale. The most important goal was to reduce the amount of time wasted on copying software from machine to machine.

## 2 First Large Scale Attempt

The package we chose as the core for this distribution system is *Track*, developed at Bell Communications Research [Nac86]. *Track* makes use of the *librarian/subscriber* control model. In this model, both the librarian and subscriber systems gather pertinent information about selected files and directories on their local disks. The librarian's information is then retrieved by a subscriber system and compared against the data that the subscriber collected. Based upon these comparisons, a subscriber is free to choose which files are to be retrieved from the librarian. Both the librarian and client have local subscription lists which determine which files are to be offered and which will be retrieved, respectively.

[1] **†UNIX is a trademark of AT&T Bell Laboratories**

Tooled with *Track*, we set out to handle all twenty servers in our department. Since all of the machines are under the same administrative domain, we soon realized that *Track* provided us with more flexibility than was needed. Specifically, we discovered that, except for minor differences, the local subscription lists on the librarian and clients were the same. To maintain these files separately was, on a small scale, reintroducing the distribution problems we were attempting to escape. We determined that if we could control the subscription lists from a central point, then we could solve their distribution problem in much the same way as we would for the system as a whole. Still, there were clients that had specific needs. Not all machines were of the same architecture, and no two machines had the same disk resources. More importantly, the actual goals of most machines differed in varying degrees. Thus, some configurability was inevitable.

To satisfy the want for central control over the subscription lists and the need for configurability, we added an additional level of indirection to our system through the use of the UNIX utility, *m4*(1). A master *m4* subscription list file would be distributed *(tracked)* to all of the subscribers and then processed along with local configuration files containing *m4* definitions. The resultant output would, in turn, be used by *Track* as the local subscription list.

The above scheme worked well enough that use of *Track* was all but mandated throughout our group. The official word from above was, "If it can be done with *Track*, it will be done with *Track*."

It was not long before our system handled more than system software updates. We began tracking more complicated files such as */etc/printcap*, */.rhosts*, and */etc/hosts.equiv*. Eventually we even began to track the */etc/group* and */etc/password* files to control administrative groups and individuals. We introduced a *motd.master* file that would allow staff to post announcements of urgency into subscriber motd files without displacing local motd information.

There were still a number of problems with our distribution scheme that were of constant concern to the members of our group:

- The transport mechanism was occasionally unreliable, and would leave systems partially installed. While this was a major problem, it was easily remedied by fixing the bugs in the *Track* software.
- System administrators would make local administrative or software changes that would be overwritten by the *Track* software. While a problem, we concluded that we could not easily affect a fix in this stage of the experiment.
- System administrators would often modify software or create temporary files on the librarian system while thinking only about the system itself and forgetting that the very same system happened to be the librarian system. These work files and partially modified programs then became candidates for tracking just like any other file in a selected directory.
- It became increasingly difficult to manipulate the master subscription list file. This file was the key to the entire distribution system, and thus, was the most fragile. Should there be a syntax error in this file, it could mean that the subscription list on all of the subscribing machines would have to be updated by hand once the problem

was determined. Worse, if a semantic error occurred, many machines could end up with unwanted software, which would generally lead to filled file systems, the worst evil.[2]

Aside from these problems, we concluded that the experiment was a major success. Several departments throughout the university had been hearing about our successes with our Track System and began to make inquiries about its use. It was now time to take stock of all that had transpired to determine what should be done to turn our Track System into a major university-wide service and resource.

## 3 A Distribution Service

System administration had already become a highly distributed task, and was seen as a local matter as machines began to appear throughout the university. Departments were already developing local expertise in their areas of interest. Systems were often jealously guarded by those who remembered the days of a central authority controlling "the" central resource. It was clear that a software distribution service could be warmly received only if it did not attempt, by accident or design, to regain control over systems out in the field. A technical solution had to recognize the importance of the following:

- system stability - The service has to deliver its goods the first time and every time. It would be all too easy to lose the trust of local system administrators if problems were to constantly arise with the distribution system itself.
- maintainability - Local administrators must find the distribution system easy to use and control. It should be possible for the distribution system to run unattended in the vast majority of applications. Logging of file transfers and arising error conditions must be complete and accurate. The base level environment parameters must be clearly delineated so as to avoid misunderstandings in the future.
- software stability - Every effort must be made to maintain the behavior of programs throughout their life cycles. Behavioral changes supporting new features should come under the control of command line options, environmental settings, or other mechanisms which might allow a program to determine what behavior is appropriate for the system on which it is currently running. If possible, compile time options should be recoded as runtime options to allow for maximum use of large software systems. Care must be taken to avoid writing policy establishing mechanisms in code where none was intended. Bug fixes which change the behavior of a program must be announced before being distributed.
- communication - A mechanism must be established that allows for an ongoing dialog between the service providers and subscribers. The importance of communication cannot be overstated. Without it, local administrators may begin to feel a loss of control as they watch changes being made to their systems without explanation. Logs which list files that have been retrieved from a librarian are not sufficient.

As is evident, none of the items above addresses the nitty-gritty details of the distribution system under discussion. However, a design for a distribution service which does not take these important concepts into consideration is most probably doomed from the start.

[2] Somehow we managed never to destroy */etc/passwd* or */etc/group*.

## 3.1 Librarian Implementation

Upon re-examination of the problems mentioned previously with regards to the goals mentioned above, we observed some interesting phenomena. First, while subscription list entries in the master *m4* file were physically grouped by directories, *m4* definitions tended to group entries throughout the file by software packages. As a consequence, the *m4* master became harder to manipulate. Second, most problems that dealt with unwanted files being tracked to a subscriber system could be attributed to using a live */usr* file system as the librarian.

The solution to the second problem was obvious. We would set aside a large area on a disk to be used as the distribution area for our new distribution system. As a consequence, we were now free to redesign the layout of the area to fit our needs and goals. To facilitate better communications between service providers and subscribers, we felt that it would be easier to speak of changes in terms of software packages rather than in terms of particular files. If we could completely automate the production of the master *m4* file, we could provide for system stability by substantially reducing the potential for catastrophic effects of human error. As with so many problems in Computer Science, we soon realized that it all boiled down to the appropriate design of our data structures; in this case, our distribution area.

The model we chose would center around the architectures of subscriber systems and the software packages that would be running on them (see figure 1). Software packages for a particular architecture type would be grouped together. An additional area was set aside to eliminate duplication of files which could be shared amongst the various architectures. A complete package would consist of a set of hierarchies resembling the UNIX file system and would be installed by applying the combination of these two hierarchies onto a subscriber system. If a package hierarchy existed solely in the area set aside for architecturally independent files, then by definition, the package would not be tied to any particular architecture and could therefore be tracked to all architectures. In order to automate the process of building master subscription lists, the software package name would uniquely identify the software package amongst the various architecture hierarchies. The software package name as a directory in a architecture hierarchy would define the root anchored positions of the directories and files found within.

To establish access-list style control over software packages, each package hierarchy would contain a special control subdirectory which would never be tracked to subscriber systems. It would contain a file which would control what systems or group of systems had access rights to the software package. The control directory would also contain a file which describes the software, in order to automate the process of building a software catalog from which administrators could choose. Along with the other files in the control directory, and judicious use of various UNIX utilities, a complete catalog entry would include the name of the software package, its availibility, size, and description. Finally, the control subdirectory would contain a file whose contents would remarkably resemble that of a *Track* subscription list. However, this file would contain only those subscription list entries needed to track the package hierarchy within the architecture tree. The processes that would then run on the librarian system became conceptually simple:

- Creating a catalog would amount to collecting the various pieces of information contained in the control directories of all of the packages and processing that information through a text processing system.

- The access control file would contain one or more keywords that would become part of the names of the master subscription lists that control distribution of a package. A package subscription list file in a control directory would be surrounded with *cpp(1)* directives. These would consist mainly of an "#ifdef _package_name_" at the beginning and an "#endif" at the end.[3] The product would be included in those master subscription lists specified in the access control file thus producing the various master subscription lists files.

- A process prepares a control subscription list for each subscriber system. The control subscription list contains entries for each master subscription list to which a particular subscriber has access.

- *Track* processes the master and control subscription list files as input and generates output which is run through a consistency checker. The consistency checker ensures that trackable objects[4] existing in the various packages do not conflict in any number of ways. Should the consistency checker identify inconsistencies in the distribution area, it disallows access to the area by subscriber systems.

Of final concern is the need for testing of software before being tracked to all subscriber systems. To provide for this, another hierarchy of directories would be created. In place of */rutgers/dist* as the root of the hierarchy (see figure 1), the new hierarchy would be named */rutgers/beta*. Its structure would be exactly the same as */rutgers/dist*. The process which builds the master subscription lists would check to see if the architecture and package it is currently processing exists in the beta distribution area. If true, the derived *cpp* construct would be somewhat more complex. Essentially, the process would build a construct which causes beta subscriber systems[5] to retrieve packages from */rutgers/beta* instead of */rutgers/dist*.

## 3.2 Subscriber Implementation

System administrators of new subscriber systems have very little to do. First, they must ensure that their systems' configurations match the base level environment parameters set by the providers of the service. Second, they must retrieve a service starter kit and a copy of the software catalog for their particular architectures and install the starter kit onto the systems. Once the service providers have enabled the new subscriber systems on the librarian machine, the systems administrators follow the instructions provided in the starter kit. These will include selecting the various packages wanted by creating a local control file consisting of *cpp(1)* "#define" directives for each package.

When a distribution system begins execution, it will first retrieve the control subscription list which was created specifically for the system in question. The control subscription list will direct the distribution system to retrieve the various master subscription lists to which the subscriber system has access. Each master subscription list will then be processed causing the selected packages to be retrieved from the librarian and placed on the subscriber's disks.

[3] In actuality, it is possible to build very complex *cpp* constructs.

[4] Trackable objects include directories, soft links and files.

[5] A beta subscriber system is one which receives all software currently under beta test. We believe that beta systems need to receive all beta test software to ensure that the software interoperates properly in a complete environment.

The last task of bringing a new system under the control of the distribution system is to decide how often to synchronize the system with the librarian and entering the appropriate entry into the local *crontab* file. Finally, system administrators will have the ongoing task of following all announcements pertaining to the distribution system and acting upon them accordingly. Such announcements will include new package announcements, bug fix announcements, and on rare occasions, instructions which should be followed to alleviate some problem with the distribution system itself.

## 4 Security

The software distribution system can provide an additional level of security to those who subscribe in a strict manner. Logs of updated files can alert administrators to intrusion of such sensitive files such as */bin/login* or */bin/sh* on both the client and librarian ends. Requesting updates on size, date, or checksum change of a file poses additional hardships to those who would supplant systems with trojan horses. However, in such a system, the weight of responsibility of security must lie on the librarian, as it is a weak link. Should that system be compromised, every subscriber would be in danger. Thus, extremely limited access should be granted to the librarian.

## 5 Future Directions

The system described in the previous sections currently serves over 250 machines located throughout Rutgers University. The cost of the current implementation is the disk space required for the staging area on the librarian system, and is directly proportional to the number of architectures supported. Excluding the cost of the initial development, maintenance of the master distribution system requires one half-time system administrator, whose time is spent inserting new and modified software into the distribution areas and announcing the updates.

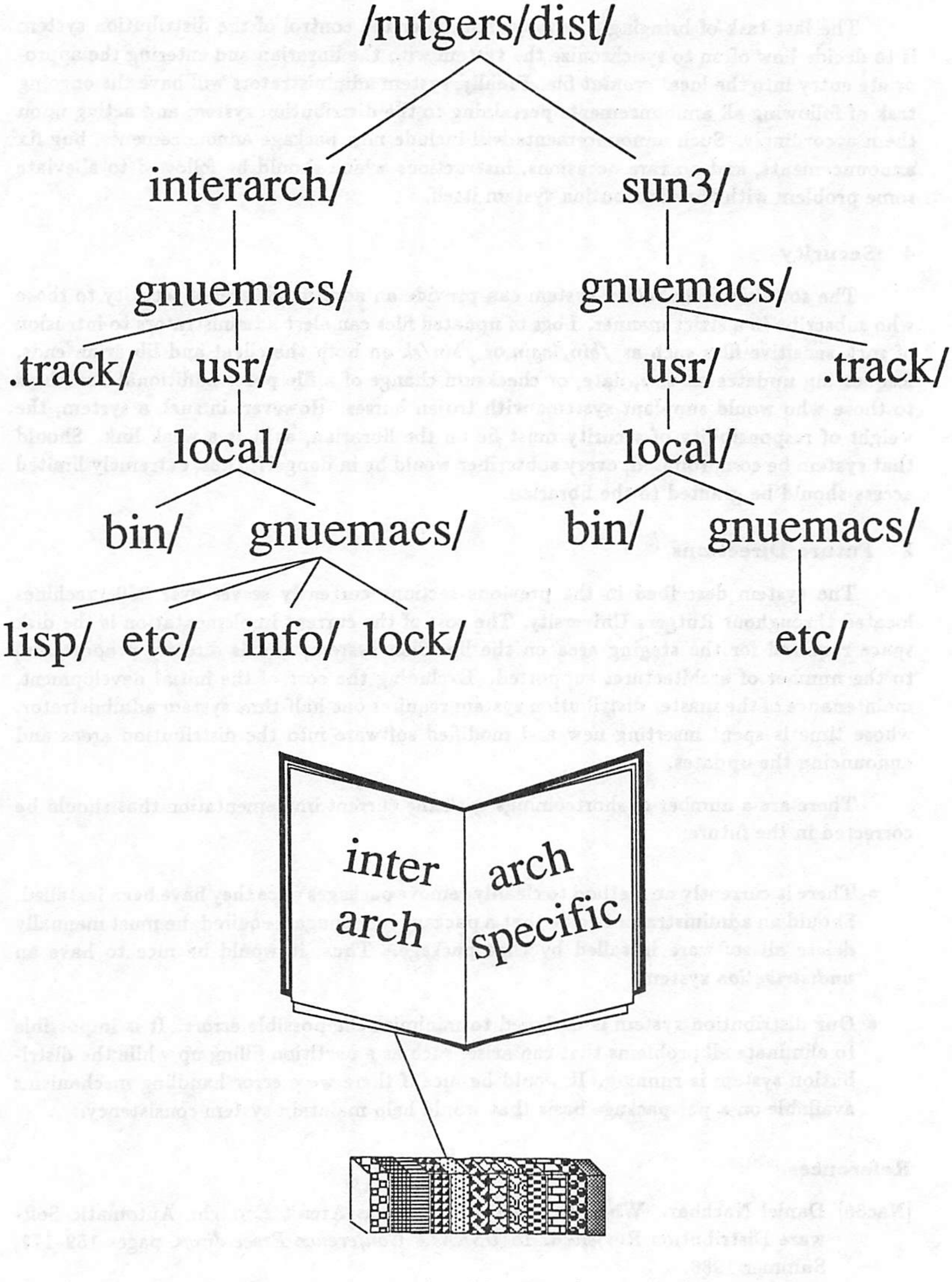
There are a number of shortcomings with the current implementation that should be corrected in the future:

- There is currently no method to cleanly remove packages once they have been installed. Should an administrator decide that a package is no longer required, he must manually delete all software installed by that package. Thus, it would be nice to have an *undistribution* system.

- Our distribution system is designed to minimize the possible errors. It is impossible to eliminate all problems that can arise, such as a partition filling up while the distribution system is running. It would be nice if there were error handling mechanisms available on a per package basis that would help maintain system consistency.

## References

[Nac86] Daniel Nachbar. When Network File Systems Aren't Enough: Automatic Software Distribution Revisited. In *USENIX Conference Proceedings*, pages 159–171, Summer 1986.



Rutgers Track II package layout

# An Empirical Study of the Reliability of Operating System Utilities

*Barton P. Miller*
*bart@cs.wisc.edu*

*Lars Fredriksen*
*fredriks@asiago.cs.wisc.edu*

*Bryan So*
*so@cs.wisc.edu*

Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

## Abstract

Operating system facilities, such as the kernel and utility programs, are assumed to be reliable. Recent experience has led us to question the robustness of our operating system utility programs under unusual input streams. Specificly, we were interested in a basic form of testing: whether the utilities would crash or hang (infinite loop) in response to unusual input. We constructed a set of tools to test this form of reliability. Almost 90 utilities were tested on each of five versions of the UNIX operating system. As a result of our tests, we were able to crash 25-32% of the utilities that we tested, including commonly used utilities such as editors, shells, and document formatters. These were programs that either terminated abnormally, generating a core dump, or programs that had infinite loops.

We then examined each utility program that crashed and identified the cause. These results are then categorized by the cause of crash. We describe the cause of the crashes, the programming practices that led to the errors, and make some suggestions on how to avoid these problems in the future.

Research supported in part by National Science Foundation grants CCR-8703373 and CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

## 1. INTRODUCTION

When we use basic operating system facilities, such as the kernel and major utility programs, we expect a high degree of reliability. These parts of the system are used frequently and this frequent use implies that the programs are well-tested and working correctly. Of course, the only way that we can insure that a program is running correctly is to use formal verification. While the technology for program verification is advancing, it has not yet reach the point where it is easy to apply (or commonly applied) to large systems.

A recent experience led us to believe that, while formal verification of a complete set of operating system utilities was too onerous task, there was still a need for some form of more complete testing. It started on a dark and stormy night. One of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. It was a race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash. These programs included a significant number of basic operating system utilities. It is reasonable to expect that basic utilities should not crash (''core dump''); on receiving unusual input, they might exit with minimal error messages, but they should not crash.

This scenario motivated a systematic test of the utility programs running on various versions of the UNIX operating system. The project proceeded in four steps: (1) construct a program to generate random characters, plus a program to help test interactive utilities; (2) use these programs to test a large number of utilities on random input strings to see if they crash; (3) identify the strings (or types of strings) that crash these programs; and (4) identify the cause of the program crashes and categorize the common mistakes that cause these crashes. As a result of testing almost 90 different utility programs on several versions of UNIX, we were able to crash more than 25% of these programs. A byproduct of this project is a list of bug reports (and fixes) for the crashed programs and a set of tools available to the systems community.

There is a rich body of research on program testing and verification. Our approach is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability. This type of study is important for several reasons. First, noisy phone lines are a reality, and major utilities (like shells and editors) should not crash because of them. Second, the bugs that caused some of the crashes were the same type of bugs that have been responsible for recent security problems [2]. We have found additional bugs that might indicate future security holes. Third, some of the crashes were caused by input that you might carelessly type. Some strange and unexpected errors were uncovered by this method of testing. Last, we sometimes inadvertently feed programs noisy input, e.g., trying to edit or view an object module. In these cases, we would like some meaningful and predictable response.

While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states. Similar techniques have been used in areas such as network protocols and CPU cache testing[†]. When testing network protocols, a module can be inserted in the data stream. This module randomly perturbs the packets (either destroying them or modifying them) to test the protocol's error detection and recovery features. Random testing has been used in evaluating complex hardware, such as a multiprocessor cache coherence protocols [3]. The state space of the device,

† There are many other areas where random testing is used. The authors welcome suggestions of specific related references.

when combined with the memory architecture, is large enough that it is difficult to generate systematic tests. Random generation of test cases can cover a large part of the state space and simplify the generation of cases.

This paper proceeds as follows. Section 2 describes the tools that we built to test the utilities. These tools include the fuzz (random character) generator, ptyjig (to test interactive utilities), and scripts to automate the testing process. Section 3 describes the tests that we performed, giving the types of input that we presented to the utilities. Results from the tests are given in Section 4, along with an analysis of the results. This analysis includes identification and classification of the program bugs that caused the crashes. Section 5 presents concluding remarks, including suggestions for avoiding the types of problems detected by our study. We include an Appendix with the user manual pages for fuzz and ptyjig.

## 2. THE TOOLS

We developed two basic programs to test the utilities. The first program, called *fuzz*, generates a stream of random characters to be consumed by a target program. There are various options to fuzz to the control the testing activity. A second program, *ptyjig*, was also written to test interactive utility programs. Interactive utilities, such as a screen editor, expect their standard input file to have the characteristics of a terminal device. In addition to these two programs, we used scripts to automate the testing of a large number of utilities.

### 2.1. Fuzz: Generating Random Input Strings

The program fuzz is basically a generator of random characters. It produces a continuous strings of characters on its standard output file (see Figure 1). We can perform different types of tests depending on the options given to fuzz. Fuzz is capable of producing both printable and control characters, only printable characters, or either of these groups along with the NULL (zero) character. You can also specify a delay between each character. This option can account for the delay in characters passing through a pipe and to help the user locate the characters that caused a utility to crash.

Fuzz can record its output stream in a file, in addition to printing to its standard output. This file can be examined later. There are options to randomly insert NEWLINE characters in the output stream, and to limit the length of the output stream. For a complete description of fuzz, see the manual page in the Appendix.

The following is an example of fuzz being used to test deqn, the equation formatter.

```
fuzz 100000 -o outfile | deqn
```

The output stream will be at most 100,000 characters in length and the stream will be recorded in file "outfile".

### 2.2. Ptyjig: Testing Interactive Utilities

There are utility programs whose input (and output) files must have the characteristics of a terminal device, e.g. the vi editor and the mail program. The standard output from fuzz sent through a pipe is not sufficient to test these programs.

Ptyjig is a program that allows us to test interactive utilities. It first allocates a pseudo-terminal file. This is a two-part device file that, on one side looks like a standard terminal device file (with a name of the form "/dev/ttyp?") and, on the other side can be used to send or receive characters on the terminal file ("/dev/ptyp?", see Figure 2). After creating the pseudo-terminal file, ptyjig then starts the specified utility program. Ptyjig passes characters that are sent to its

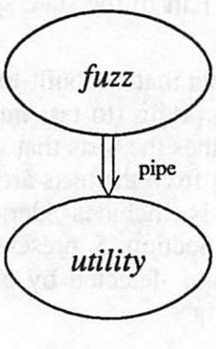


**Figure 1: Output of fuzz piped to a utility.**

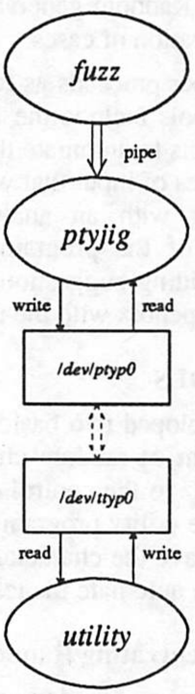


**Figure 2: Fuzz with ptyjig to test an interactive utility.**

*/dev/ttyp0 is a pseudo-terminal device and /dev/ptyp0 is a pseudo-device to control the terminal.*

input through the pseudo-terminal to be read by the utility.

The following is an example of fuzz and ptyjig being used to test vi, a terminal-based screen editor.

```
fuzz 100000 -o outfile | ptyjig vi
```

The output stream of fuzz will be at most 100,000 characters in length and the stream will be recorded in file ''output''. For a complete description of ptyjig, see the manual page in the Appendix.

### 2.3. The Scripts: Automating the Tests

A command (shell) script file was written for each type of test. Each script executes all of the utilities for a given set of input characteristics. The script checks for the existence of a ''core'' file after each utility terminates; indicating the crash of that utility. The core file and the offending input data file are saved for later analysis.

## 3. THE TESTS

After building the software tools, we then used these tools to test a large collection of utilities running on several versions of the UNIX operating system. Each utility on each system was executed with several different types of input streams. A test of a utility program can produce one of three results: (1) *crash* – the program terminated abnormally producing a core file, (2) *hang* – the program appeared to loop indefinitely, or (3) *succeed* – the program terminated normally. Note that in the last case, we do not specify the correctness of the output.

To date, we have tested utilities on five versions of UNIX†. These versions are summarized in Table 1. Most of these versions are derived from some form of 4.2BSD or 4.3BSD Berkeley UNIX. Some versions, like the SunOS release, have undergone substantial revision (especially at the kernel level). The SCO Xenix version is based on the System V standard from AT&T. It is also important that the tests covered several hardware architectures, as well as several systems. A program statement with an error might be tolerated on one machine and cause the program to crash on another. Referencing through a null-value pointer is an example of this type of problem.

Our testing covered a total of 88 utility programs on the five versions of UNIX. Most utilities were tested on each system. Table 2 lists the names of the utilities that were tested, along with the type of each system on which that utility was tested. For a detailed description of each of these utilities, we refer you to the user manual for appropriate systems. The list of utilities covers a substantial part of those that are commonly used, such as the mail program, screen editors, compilers, and document formatting packages. The list also includes less commonly used utilities, such as cb, the C language pretty-printer.

Each utility program that we tested was subjected to several different types of input streams. The different types of inputs were intended to test for a variety of errors that might be triggered in the utilities that we were testing. The major variations in test data were including non-printable (control) characters, including the NULL (zero) byte, and maximum length of the input stream. These tests are summarized in Table 3a.

| **Versions of UNIX Test** | | | |
|---|---|---|---|
| **Identifying Letter** | **Machine Vendor** | **Processor** | **Kernel** |
| v | DEC Vaxstation 3200 | CVAX | 4.3BSD + NSF (from Wisconsin) |
| s | Sun 4/110 | SPARC | SunOS 3.2 & SunOS 4.0 with NSF |
| h | HP Bobcat 9000/320<br>HP Bobcat 9000/330 | 68020<br>68030 | 4.3BSD + NSF (from Wisconsin), with Sys V shared-memory |
| x | Citrus 80386 | i386 | SCO Xenix System V Rel. 2.3.1 |
| r | IBM RT/PC | ROMP | AOS UNIX |

**Table 1: List of Systems Tested**

† Only the csh utility was tested on the IBM RT/PC. More complete testing is in progress.

| Utility | VAX | Sun | HP | i386 |
|---|---|---|---|---|
| adb | ● ○ | ● | ● | ○ |
| as | ● | | | ● |
| awk | | | | |
| bc | | | | ● ○ |
| bib | | | – | – |
| calendar | | | | – |
| cat | | | | |
| cb | ● | | ● | ● |
| cc | | | | |
| /lib/ccom | | | | – |
| checkeq | | | | – |
| checknr | | | | – |
| col | ● ○ | ● | ● | ● ○ |
| colcrt | | | | – |
| colrm | | | | – |
| comm | | | | |
| compress | | | | |
| /lib/cpp | | | | |
| csh | ● ○ | ○ | ○ | |
| dbx | | * | – | – |
| dc | | | | ○ |
| deqn | | ● | – | – |
| deroff | ● | ● | ● | |
| diction | ● | – | ● | |
| diff | | | | |
| ditroff | ● ○ | ● | – | – |
| dtbl | | | – | – |
| emacs | ● | ● | ○ | – |
| eqn | | ● | ● | ● |
| expand | | | | |
| f77 | ● | | – | – |
| fmt | | | | |
| fold | | | | |
| ftp | ● | ● | ● | – |
| graph | | | | |
| grep | | | | |
| grn | | | – | – |
| head | | | | |
| ideal | | | – | – |
| indent | ● ○ | ● ○ | ● | – |
| join | | ● | | |
| latex | | | – | – |
| lex | ● | ● | ● | ● |
| lint | | | | |
| lisp | | – | | – |
| look | ● | ○ | ● | ● |

**Table 2: List of Utilities Tested and the Systems on which They Were Tested (part 1)**

*● = utility crashed, ○ = utility hung, * = crashed on SunOS 3.2 but not on SunOS 4.0,*
*– = utility unavailable on that system. N.B. join crashed only on SunOS 4.0, not 3.2.*

| Utility | VAX | Sun | HP | i386 |
|---|---|---|---|---|
| m4 | | | | ● |
| mail | | | | |
| make | | | ● | |
| more | | | | |
| nm | | | | |
| nroff | | | | ● |
| pc | | | | – |
| pic | | | – | – |
| plot | – | ○ | | – |
| pr | | | | |
| prolog | ● ○ | ● ○ | ● ○ | – |
| psdit | | | | – |
| ptx | – | ● | ● | ○ |
| refer | ● | * | ● | – |
| rev | | | | – |
| sed | | | | |
| sh | | | | – |
| soelim | | | | |
| sort | | | | |
| spell | ● ○ | ● | ● | ○ |
| spline | | | | |
| split | | | | |
| sql | | – | | |
| strings | | | | |
| strip | | | | |
| style | ● | – | ● | |
| sum | | | | |
| tail | | | | |
| tbl | | | | |
| tee | | | | |
| telnet | ● | ● | ● | – |
| tex | | | – | – |
| tr | | | | |
| troff | – | – | – | |
| tsort | ● | * | ● | ● |
| ul | ● | ● | ● | – |
| uniq | ● | ● | ● | ● |
| units | ● ○ | ● | ● | ● |
| vgrind | ● | | – | – |
| vi | ● | | ● | |
| wc | | | | |
| yacc | | | | |
| # tested | 85 | 83 | 75 | 57 |
| # crashed/locked | 25 | 21 | 24 | 16 |
| % | 29.4% | 25.3% | 32.0% | 28.1% |

**Table 2: List of Utilities Tested and the Systems on which They Were Tested (part 2)**

● = *utility crashed,* ○ = *utility hung,* * = *crashed on SunOS 3.2 but not on SunOS 4.0,*
– = *utility unavailable on that system. N.B. join crashed only on SunOS 4.0, not 3.2.*

The input streams for interactive utilities have slightly different characteristics. To avoid overflowing the input buffers on the terminal device, the input was split into random length lines (i.e., terminated by a NEWLINE character) with a mean length of 128 characters. The input length parameter is described in number of lines, so is scaled down by a factor of 100.

## 4. THE RESULTS AND ANALYSIS

Our tests of the UNIX utilities produced a surprising number of programs that would crash or hang. In this section, we summarize these results, group the results by the common programming errors that caused the crashes, and show the programming practices that caused the errors.

| Input Streams for Non-Interactive Utilities | | | |
|---|---|---|---|
| # | Character Types | NULL character | Input stream size (no. of bytes) |
| 1 | printable+nonprintable | • | 1000 |
| 2 | printable+nonprintable | • | 10000 |
| 3 | printable+nonprintable | • | 100000 |
| 4 | printable | • | 1000 |
| 5 | printable | • | 10000 |
| 6 | printable | • | 100000 |
| 7 | printable+nonprintable | | 1000 |
| 8 | printable+nonprintable | | 10000 |
| 9 | printable+nonprintable | | 100000 |
| 10 | printable | | 1000 |
| 11 | printable | | 10000 |
| 12 | printable | | 100000 |

**Table 3a: Variations of Input Data Streams for Testing Utilities**
*(these were used for the non-interactive utility programs)*

| Input Streams for Interactive Utilities | | | |
|---|---|---|---|
| # | Character Types | NULL character | Input stream size (no. of strings) |
| 1 | printable+nonprintable | • | 10 |
| 2 | printable+nonprintable | • | 100 |
| 3 | printable+nonprintable | • | 1000 |
| 4 | printable | • | 10 |
| 5 | printable | • | 100 |
| 6 | printable | • | 1000 |
| 7 | printable+nonprintable | | 10 |
| 8 | printable+nonprintable | | 100 |
| 9 | printable+nonprintable | | 1000 |
| 10 | printable | | 10 |
| 11 | printable | | 100 |
| 12 | printable | | 1000 |

**Table 3b: Variations of Input Data Streams for Testing Utilities**
*(these were used for the interactive utility programs)*

The basic test results are summarized in Table 2. The first result to notice is that we were able to crash or hang a significant number of utility programs on each system (from 25-32%). Included in the list of programs are several commonly used utilities, such as: vi and emacs, the most popular screen editors; csh, the c-shell; and various programs for document formatting. We detected two types of error results, crashing and hanging. A program was considered crashed if it terminated producing a core (state dump) file, and was considered hung if it continued executing producing no output while having available input. A program was also considered hung if it continued to produce output after its input had stopped. Hung programs were typically allowed to execute for an additional five minutes after the hung state was detected.

Table 4 summarizes the list of utility programs that we were able to crash or hang, categorized by the cause of the crash, and showing on which systems we were able to crash the programs. Notice that a utility might crash on one system but not on another. This result is due to several reasons. One reason is differences in the processor architecture. For example, while the VAX will (incorrectly) tolerate references through null pointers, many other architectures will not (e.g., the Sun 4). A second reason is that the different systems had differences in the versions of the utilities. Local changes might improve or degrade a utility's reliability. Both internal structure as well as external specification of the utilities change from system to system.

We grouped the causes of the crashes into the following categories: pointer/array errors, not checking return codes, input functions, sub-processes, interaction effects, bad error handler, signed characters, race conditions, and currently undetermined. For each of these categories, we discuss the error, show code fragments as examples of the error, present implications of the error, and suggest fixes for the problem.

*Pointer/Arrays*

The first class of pointer and array errors is the case where a program might sequentially access the cells of an array with a pointer or array subscript, while not checking for exceeding the range of the array. This was one of the most common programming errors found in our tests. An example (taken from cb) shows this error using character input:

```
while ((cc = getch()) != c){
     string[j++] = cc;
     . . .
}
```

The above example could be easily fixed to check for a maximum array length. Often the terseness of the C programming style is carried to extremes; form is emphasized over correct function. The ability to overflow an input buffer is also a potential security hole, as shown by the recent Internet worm.

The second class of pointer problems is caused by references through a null pointer. The prolog interpreter, in its main loop, can incorrectly set a pointer value that is then assumed to be valid in the next pass around the loop. A crash caused by this type of error can occur in one of two places. On machines like the VAX, the reference through the null pointer is valid and reads data at location zero. The data accessed are machine instructions. A field in the (incorrectly) accessed data is then used as a pointer and the crash occurs. On machines like the Sun 4, the reference through the null pointer is an error and crashes immediately. If the path from where the pointer was set to where it was used is not an obvious one, extra checking may be needed.

The assembly language debugger (adb) also had a reference through a null pointer. In this case, the pointer was supposed to be a global variable that was set in another module. The external (global) definition was accidentally omitted from the variable declaration in the module that expected to use the pointer. This module then referenced an uninitialized (in UNIX, zero)

| | Cause | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Utility | array/ pointer | NCRC | input functions | sub-processes | interaction effects | bad error handler | signed characters | race condition | no source code | unknown |
| adb | vshx | v | | | | | | | | |
| as | vx | | | | | | | | | |
| bc | | | | | | | | | x | |
| cb | vhx | | | | | | | | | |
| col | | vshx | | | | | | | | |
| csh | | | | | vshr | | | vshr | | |
| dc | | | | | | | | | x | |
| deqn | | | | | | | s | | | |
| deroff | vsh | | | | | | | | | |
| diction | | | | vh | | | | | | |
| ditroff | vs | | | | | | | | | |
| emacs | | | | | | | | vsh | | |
| eqn | | | | | | | shx | | | |
| f77 | | | | | | | | | | v |
| ftp | | | vsh | | | | | | | |
| indent | vsh | | | | | | | | | |
| join | | | | | | | | | s | |
| lex | vshx | | | | | | | | | |
| look | vshx | | | | | | | | | |
| m4 | | | | | | | | | x | |
| make | h | | | | | | | | | |
| nroff | | | | | | | | | x | |
| plot | | | | | | | | | | s |
| prolog | vsh | | | | | | | | | |
| ptx | shx | | | | | | | | | |
| refer | vsh | | | | | | | | | |
| spell | vshx | | | | | | | | | |
| style | | | | vh | | | | | | |
| telnet | | | vsh | | | | | | | |
| tsort | | | vshx | | | | | | | |
| ul | vsh | | | | | | | | | |
| uniq | vshx | | | | | | | | | |
| units | vshx | | | | | v | | | | |
| vgrind | | | | v | | | | | | |
| vi | | | | vh | | | | | | |

**Table 4: List of Utilities that Crashed, Categorized by Cause**

*The letters indicate the system on which the crash occurred (see Table 1).*

pointer.

Pointer errors do not always appear as bad references. A pointer might contain a bad address that, when used to write a variable, may unintentionally overwrite some other data or code location. It is then unpredictable when the error will manifest itself. In our tests, the crash of lex (scanner generator) and ptx (permuted index generator) were examples of overwriting data, and the crash of ul (underlining text) was an example of overwriting code.

The crash of as (the assembler) originally appeared to be a result of improper use of an input routine. The crash occurred at a call to the standard input library routine ungetc(), which returns a character back to the input buffer (often used for look ahead processing). The actual cause was that ungetc() was redefined in the program as a macro that did a similar function. Unfortunately, the new macro had less error checking than the system version of ungetc() and allowed a buffer pointer to be incorrectly set. Since the new macro looks like the original routine, it is easy to forget the differences.

*Not Checking Return Codes*

Not checking return codes is a sign of careless programming. It is a favorable comment on the current state of UNIX that there are so few examples of this error. During our tests, we were able to crash adb (the assembly language debugger) and col (multi-column output filter ASCII terminals) utilities crash due to this error. Adb provides an interesting example of a programming practice to avoid. This code fragment represents the a loop in adb and a procedure called from that loop.

```
format.c (line 276):

. . .
while (lastc != '\n') {
          rdc();
}
. . .


input.c (line 27):

rdc()
{    do { readchar(); }
     while (lastc == ' ' || lastc== '\t');
     return (lastc);
}
```

The initial loop reads characters, one by one, terminating when the end of a line has been seen. The rdc() routine calls readchar(), which places the new character into a global variable named "lastc". Rdc() will skip over tab and space characters. Readchar() uses the UNIX file read kernel call to read the characters. If readchar() detects the end of the input file, it will set the value of lastc to zero. Neither rdc() nor the initial loop check for the end of file. If the end of file is detected during the middle of a line, this program hangs.

We can speculate as to why there was no end of file check on the initial loop. It may be because the program author thought it unlikely that the end of file would occur in this situation. It might also be that it was awkward to handle the end of file in this location. While this is not difficult to program, it requires extra tests and flags, more complex loop conditions, or possibly the use of a goto statement.

This problem was made more complex to diagnose due to the extensive use of macros (the code fragment above has the macros expanded). These macros may have made it easier to overlook the need for the extra test for end of file.

*Input Functions*

We have already seen cases where character input routines within a loop can cause a program to store into locations past the end of an array. Input routines that read entire strings are also vulnerable. One of the main holes through which the Internet worm entered was the gets()

routine. The gets() routine takes a single parameter that is a pointer to a character string. No means of bounds checking are possible. Our tests crashed the ftp and telnet utilities through use of gets().

The scanf() routine is also vulnerable. In the input specification, it is possible to specify an unbounded string field. An example of this comes from the tsort (topological sort) utility.

```
        x = fscanf(input,"%s%s",precedes, follows);
```

The input format field specifies two, unbounded strings. In the program, ''proceeds'' and ''follows'' are declared with the relatively small lengths of 50 characters. It is possible to place a bound on the string field specification, solving this problem.

*Sub-Processes*

The code you write might be carefully designed and written and you might follow all the good rules for writing programs. But this might not be enough if you make use of another program. as part of your program. Several of the UNIX utilities execute other utilities as part of doing their work. For example, the diction and style utilities call deroff, vi calls csh, and vgrind calls troff. When these sub-processes are called, they are often given direct access to the raw input data stream, so they are vulnerable to erroneous input. Access to sub-processes should be carefully controlled or you should insure that the program input to the sub-process is first checked. Alternatively, the utility should be programmed to tolerate the failure of a sub-process (though, this can be difficult).

*Interaction Effects*

Perhaps one of the most interesting errors that we discovered was a result of an unusual interaction of two parts of csh, along with a little careless programming. The following string will cause the VAX version of csh to crash

```
!a%8f
```

and the following string

```
!a%888888888f
```

will hang (continuous output of space characters) most versions of csh. The first example triggers the csh's command history mechanism, says ''repeat the last command that began with 'a%8f' ''. Since it does not find such a command, csh forms an error message string of the form: ''a%8f: Event not found.'' This string is passed to the error printing routine, which uses the string as the first parameter to the printf() function. The first parameter to printf() can include format items, denoted by a ''%''. The ''%8f'' describes a floating point value printed in a field that is 8 characters wide. Each format item expects an additional parameter to printf(), but in the csh error, none is supplied (or expected).

The second example string follows the same path, but causes csh to try to print the floating point value in a field that is 888,888,888 characters wide. The (seemingly) infinite loop is the printf() routine's attempt to pad the output field with sufficient leading space characters. Both of these errors could be prevented by substituting the printf() call with a simple string printing routine (such as puts()). The printf() was used for historical reasons having to do with space efficiency. The error printing routine assumed that it would always be passed strings that were safe to print.

*Bad Error Handler*

Sometimes the best intentions do not reach completion. The units program detects and traps floating point arithmetic errors. Unfortunately, the error recovery routine only increments a count of the number of errors detected. When control is returned to the faulty code, the error recurs, resulting in an infinite loop.

*Signed Characters*

The ASCII character code is design so that codes normally fall in the range that can be represented in seven bits. The equation formatter (eqn) depends on this assumption. Characters are read into an array of signed 8-bit integers (the default of signed vs. unsigned characters in C varies from compiler to compiler). These characters are then used to compute a hash function. If an 8-bit character value is read, it will appear as a negative number and result in an erroneous hash value. The index to the hash table will then be out of range. This problem can be easily fixed by using unsigned values for the character buffer. In a more sophisticated language than C, characters and strings would be identified as a specific type not related to integers.

This error does not crash all versions of adb. The consequence of the error depends on where in the address space is accessed by the bad hash value.

*Race Conditions*

UNIX provides a signal mechanism to allow a program to asynchronously respond to unusual events. These events include keyboard-selected functions to kill the program (usually control-C), kill the program with a core dump (usually control-\), and suspend the program (usually control-Z). There are some programs that do not want to allow themselves to be interrupted or suspended; they want to process these control characters directly, perhaps taking some intermediate action before terminating or suspending themselves. Programs that make use of the cursor motions features of a terminal are examples of programs that directly process these special characters. When these programs start executing, they place the terminal device in a state that overrides processing of the special characters. When these programs exit, it is important that they restore the device to its original state.

So, when a program, such as emacs, receives the suspend character, it appears as an ordinary control-Z character (not triggering the suspend signal). Emacs will, on reading a control-Z, do the following: (1) reset the terminal to its original state (and will now respond to suspend or terminate signals), (2) clean up its internal data structures, and (3) generate a suspend signal to let the kernel actually stop the program.

If a control-\ character is received on input between steps (1) and (3), then the program will terminate, generating a core dump. This race condition is inherent in the UNIX signal mechanism since a process cannot reset the terminal and exit in one atomic operation. Other programs, such as vi and more, are also subject to the same problem. The problem is less likely in these other programs because they do less processing between steps (1) and (3), providing a smaller window of vulnerability.

*Undetermined Errors*

The last two columns of Table 4 list the programs where the source code was currently not available to us or where we have not yet determined the cause of the crash.

## 5. CONCLUSIONS

This project started as a simple experiment to try to better understand an observed phenomenon – that of programs crashing when we used a noisy dial-up line. As a result of testing a comprehensive list of utility programs on several versions of UNIX, it appears that this is not an isolated problem. We offer two tangible products as a result of this project. First, we provide a list of bug reports to fix the utilities that we were able to crash. This should make a qualitative improvement on that reliability of UNIX utilities. Second, we provide a test method (and tools) that is simple to use, yet surprisingly effective.

We do not claim that our tests are exhaustive; formal verification is required to make such strong claims. We cannot even estimate how many bugs are still yet to be found in a given program. But our simple testing technique has discovered a wealth of errors and is likely to be more commonly used (at least in the near term) than more formal procedures.

Our examination of the results of the tests have exposed several common mistakes made by programmers. Most of these mistakes are things that experienced programmers already know, but an occasional reminder is sometimes helpful. From our inspection of the errors that we have found, following are some suggested guidelines:

(1) All array references should be checked for valid bounds. This is an argument for using range checking full-time. Even (especially!) pointer-based array references in C should be checked. This spoils the terse and elegant style often used by experienced C programmers, but correct programs are more elegant than incorrect ones.

(2) All input fields should be bounded – this is just an extension of guideline (1). In UNIX, using ``%s'' without a length specification in an input format is bad idea.

(3) Check all system call return values; do this checking even when a error result is unlikely and the response to a error result is awkward.

(4) Pointer values should often be checked before being used. If all the paths to a reference are not obvious, an extra sanity check can help catch unexpected problems.

(5) Judiciously extend your trust to others; they may not be as careful a programmer as you. If you have to use someone else's program, make sure that the data you feed it has been checked.

(6) If you redefine something to look too much like something else, you may eventually forget about the redefinition. You then become subject to problems that occur due to the hidden differences. This may be an argument against excessive use of procedure overloading in languages such as Ada or C++.

(7) Error handlers should handle errors. These routines should be thoroughly tested so that they do not introduce new errors or obfuscate old ones.

(8) Goto statements are generally a bad idea. Dijskstra observed this many years ago [1], but some programmers are difficult to convince. Our search for the cause of a bad pointer in the prolog interpreter's main loop was complicated by the interesting weaving of control flow caused by the goto statements.

We still have many experiments left to perform. We have tested only the utilities that are directly accessible by the user. Network services should also receive the same attention. It is a simple matter to construct a ``portjig'' program, analogous to our ptyjig, to allow us to connect to a network service and feed it the output of the fuzz generator. A second area to examine is the processing of command line parameters to utilities. Again, it would be simple to construct a ``parmjig'' that would start up utilities with the command line parameters being generated by the randoms strings from the fuzz generator. A third area is to study other operating systems. While UNIX pipes make it simple to apply our techniques, utility programs can still be tested on other systems. The random strings from fuzz can be placed a in file and the file used as program input. A comparison across different systems would provide a more comprehensive statement on operating system reliability.

Our next step is to fix the bugs that we have found and re-apply our tests. This re-testing may discover new program errors that were masked by the errors found in the first study. We believe that a few of rounds of testing will be needed before we reach the limits of our tools.

We are making our testing tools generally available and invite others to duplicate and extend our tests.

**ACKNOWLEDGEMENTS**

We are grateful to Matt Thurmaier of Breakpoint Computer Systems (Madison) for providing us with use of the Citrus 386-based XENIX machine. Also, our thanks to Dave Cohrs for his help in locating the race condition that caused emacs to crash.

## 6. REFERENCES

[1] E. W. Dijkstra, ''GOTO Statement Considered Harmful,'' *Communications of the ACM* **11**(3) pp. 147-8 (March 1968).

[2] D. Seeley, ''A Tour of the Worm,'' *Proc. of the 1989 Winter USENIX Technical Conf.*, pp. 287-304 (January 1989).

[3] D. A. Wood, G. A. Gibson, and R. H. Katz, ''Verifying a Multiprocessor Cache Controller Using Random Case Generation,'' Computer Science Technical Report UCB/CSD 89/490, University of California, Berkeley (January 1989).

**NAME**

fuzz – random character generator

**SYNOPSIS**

**fuzz** length [ option ] ...

**DESCRIPTION**

The main purpose of *fuzz* is to test the robustness of system utilities. We use *fuzz* to generate random characters. These are then piped to a system utility (using *pty(1)* if necessary.) If the utility crashes, the saved input and output streams can then be analyzed to decide what sorts of input cause problems.

*Length* is taken to be the length of the output stream, usually in bytes, When **–l** is selected it the length is in number of strings.

The following options can be specified.

**–0** Include NULL (ASCII 0) characters

**–a** Include all ASCII characters except NULL (default)

**–d** *delay*
Specify a delay in seconds between each character.

**–e** *string*
Send *string* after all the characters. This feature can be used to send termination strings to the test programs. Standard C escape sequences can be used.

**–l** *[len]*
Generate random length strings. If *len* is specified, it is taken to be the maximum length of each string (default = 255). Strings are terminated with the ASCII newline character.

**–o** *file* Store the output stream to *file* as well as sending them to *stdout.*

**–p** Generate printable ASCII characters only

**–r** *file* Replay characters stored in *file.*

**–s** *seed* Use *seed* as the seed to the random number generator.

**–x** Print the seed as the first line of stdout.

**AUTHORS**

Lars Fredriksen, Bryan So.

**SEE ALSO**

pty(1)

**NAME**

ptyjig – pseudo-terminal pipe

**SYNOPSIS**

**ptyjig** [ option ] ... command [ args ] ...

**DESCRIPTION**

*Pty* executes the Unix *command* with *args* as its arguments if supplied. The standard input of ptyjig is piped to *command* as if typed at a terminal. Ptyjib is expected to be used with *fuzz(1)* to test interactive (terminal based) programs.

The following options can be specified.

**–e** Do not send EOF character after *stdin* has exhausted.

**–s** Do not process interrupt signals, such as SIGINT, SIGQUIT and SIGSTOP.

**–x** Do not write output from *command* to *stdout.*

**–i** *file* Save the input stream sent to *command* into *file.*

**–o** *file* Save the output produced by *command* into *file.*

**–d** *delay*
Wait *delay* seconds after sending each character.

**–t** *interval*
If input has exhausted but *command* has neither exited nor sent any output, exit after *interval* seconds. Default is 2.0 seconds.

*Delay* and *interval* can have fractions.

**EXAMPLE**

```
ptyjig -o out -d 0.2 -t 10 vi text1 <text2
```

Runs "vi text1" in background, typing the characters in *text2* into it with a delay of 0.2sec between characters, and save the output to *out.* The program stops when *vi* stops outputting for 10 seconds.

**AUTHORS**

Lars Fredriksen, Bryan So.

**FILES**

/dev/tty*
/dev/pty*

**SEE ALSO**

fuzz(1), sigvec(2), pty(4), tty(4)

**BUGS**

The trace files specified by –i and –o options may contain more than actual characters sent to and received from *command.* This is due to the fact that after *command* exits and before *ptyjig* is signaled, some characters may be sent. This can be prevented by setting –d option to some suitable delay.

If the test program terminates abnormally, the usual core dumped message is not printed.

# RAPID: Remote Automated Patch Installation Database

*Russell Brand*
Lawrence Livermore National Laboratory
Livermore, CA 94550
brand@lll-crg.llnl.gov

*D. Brent Chapman*
Chapman Consulting
6 Schooner Court
Richmond, CA 94804
brent@lll-crg.llnl.gov

## 1. Executive Summary

Lawrence Livermore National Labs (LLNL) currently operates over a thousand unclassified computer systems that are accessible to the outside world; these systems use more than twenty different hardware and operating system combinations. Responsibility for the day-to-day operation of these machines rests with hundreds of different people. While all these machines *should* be secured by the timely and routine installation of security patches (as the patches are developed), it is unrealistic to expect *all* of the hundreds of responsible individuals to install *all* of the necessary patches in a timely manner.

This is a continuing problem; new hardware and software systems almost always introduce new bugs and security problems, and additional problems are continually being found in existing systems. RAPID (Remote Automated Patch Installation Database) will facilitate the easy and timely creation, distribution, and installation of software patches (primarily security-oriented, but also bugfix-oriented) throughout LLNL's unclassified computer systems. It will also provide LLNL with a secure authentication service (to ensure that a user is really who he or she claims to be), tools for improved general software maintenance, and greater administrative control over laboratory computing.

## 2. Background

LLNL physically and electronically separates its classified computing facilities from the outside world in order to ensure the security of those facilities. Unfortunately, this approach is impractical for LLNL's unclassified systems, since they *must* allow authorized access to the outside world in order to fulfill their missions. LLNL currently has more than a thousand machines, using more than twenty hardware and operating system combinations, that are accessible to the outside world through high-speed networks or dialin lines. Most of these machines have known, correctable security holes. The problem is that these machines are controlled by hundreds of different individuals, and it is extremely difficult to both contact all the appropriate individuals concerning security problems and patches *and* to ensure that all of the patches are installed properly (if at all) by all of the individuals.

Over the past few months, in response to several computer security incidents, it has been necessary to create and install security-related software patches on hundreds of machines at LLNL. Because LLNL lacks an automated system for this purpose, administrative and gate memos (memos handed to people by security staff as the arrive at the gate each morning) have been used to instruct system managers to perform the long, complex tasks involved to install the patches. The costs of this method are great, both in terms of diverted manpower and negative publicity. Further, not all of the patches have been installed by all of the system managers, meaning that some systems are still vulnerable to known, correctable problems.

A case in point is the InterNet worm incident [Spafford88, Eichin88, NSANCSC88]. In late November 1988, a "worm" was released on the the InterNet from a university computer system at Cornell. The worm entered tens of thousands of computers nationwide by exploiting known security problems. The effects on LLNL and the Department of Energy were limited due to the swift reaction of computer security and system programming staff, but even now it is still unknown exactly which LLNL systems were affected, nor have more than a small number of the potentially vulnerable systems received the necessary security patches to close the holes that the worm exploited.

A second exemplary case is the recent TIS breakin. In late November 1988, a cracker entered a number of computers operated by the TIS project at LLNL. TIS system staff discovered the breakins in early December when the cracker damaged key system files. Efforts of LLNL system programmers and computer security staff revealed that many LLNL machines had been entered by the cracker, both in the TIS project and elsewhere in the lab. Many of the unauthorized accesses to LLNL machines had occurred from other InterNet sites throughout the world, and LLNL machines had been used to attack still more InterNet machines. Working with the national Computer Emergency Response Team, LLNL staff notified the affected sites that had been attacked and infiltrated; none of these sites had been aware that they had been compromised. At several of these sites, operating system software had been modified by the crackers to enable them to more effectively attack other machines; these modifications mean that the managers of the affected systems must reload their entire operating systems from backups done *before* their machines were compromised, and then try to recreate legitimate changes that were made between the time of the backup and the discovery of the crackers. This is generally a long and painful process requiring the efforts of a number of system people for a number of days on each affected system.

Similar incidents have occurred at many other sites, including Stanford University [Reid87] and Lawrence Berkeley Laboratories [Stoll88]. In each of these incidents, *known* security problems were exploited. If the sites affected had been operating a RAPID-like system, these known problems would already have been corrected, and the incidents would never have taken place (or would at least have been less damaging).

## 3. Goals, Tasks, and Milestones

LLNL currently remains vulnerable for three reasons. First, it is often hard to create, test, and install security patches (especially for systems where operating system source code is unavailable). Second, it is hard to repeatedly get the attention and timely cooperation of the hundreds of system managers at LLNL, currently the people who must install the patches. Third, security problems have historicly demonstrated a disturbing tendency to be fixed only to reappear in similar or identical form when a system is reconstructed from backups or when a new operating system release is installed; not only must the problems be fixed, but someone has to ensure that they *stay* fixed. Rather than continuing to rely on the uncertain cooperation of these hundreds of system managers (using ineffective communication methods that are likely to gain undesired press attention), RAPID will be an automated system for installing security and bugfix patches on all unclassified computers at LLNL.

Initially, RAPID will service only LLNL's most common operating systems, but will later expand to cover all unclassified systems. It is estimated that a handful of operating systems (UNIX from Sun, DEC, Berkeley, and AT&T, and VMS from DEC) account for the majority of unclassified systems at LLNL. Once the development phase of RAPID is completed, the system can be replicated throughout the Department of Energy.

RAPID will have several tasks:

A. Identify security problems and bugs in LLNL systems
B. Create patches for those problems and bugs for LLNL systems
C. Install those patches on LLNL systems
D. Interact with cooperating groups outside LLNL
E. Map LLNL's computer network, as it stands now and as it changes and grows
F. Provide a secure authentication mechanism for LLNL systems
G. Provide a reliable communication resource
H. Produce a plan to provide RAPID services to VMS systems
I. Produce a plan to provide RAPID services to classified systems

RAPID will not do active monitoring for intruders, nor will will it be designed as a crisis response center. By its nature, however, it will sometimes be the first to detect problems, and it will of course assist in quickly creating and installing security patches throughout LLNL's systems when emergencies *do* arise (as they inevitably will). RAPID's primary mission will be to *prevent* emergencies, not necessarily to respond to them. It should not be necessary to provide 24-hour "hotlines" to RAPID, nor will RAPID's staff or "dedicated machines" routinely host an emergency response group. While some members of the RAPID staff my participate in Emergency Response, and part of RAPID's development resources could be made available during a crisis, it is essential to RAPID's success that it and its "dedicated machines" be able to run as a stable, trusted production facility.

RAPID will not be engaging in research per se. The authentication scheme will be based on Kerberos from MIT, which appears destined to become an industry standard [Miller87, Neuman88, Steiner88]. The patch installation system will be based primarily on Track, with enhancements developed at Rutgers University and adaptations from MIT [Nachbar86, Harrison88, Truscott86, Lord88]. Both of these packages are available without cost, and the developers of the systems are willing to provide assistance as needed; both of these packages have been running successfully on networks similar in size and scope to LLNL's for several years.

## 3.1. Identifying Security Problems and Bugs

There are a number of forums where computer security bugs and fixes are discussed, including InterNet mailing lists, technical conferences, net-news, and bboards. RAPID staff will monitor and take part in these forums, and will make other arrangements (for instance, with the Computer Systems Research Group at Berkeley and the Computer Emergency Response Team at Carnegie-Mellon) to assure that LLNL promptly learns about security problems and their solutions.

This task will be permanently ongoing, regardless of the state of the rest of RAPID's activities. Because of its ongoing nature, the only significant milestones will be the identification of and subscription to all known forums for computer security discussions, and the establishment of an archiving and indexing scheme for all the information that will be collected.

## 3.2. Creating Patches for LLNL Systems

Creating security and bugfix patches is generally easy only when source code is available. In most cases, LLNL already has the necessary code, but in some cases where source code is not available, it may be necessary to disable certain services on certain machines for a limited time until other methods of fixing the problems can be employed. Once the RAPID system is out of the development phase, the bulk of the RAPID staff's work will be creating patches.

This is another ongoing task, for which it is difficult to identify milestones. Development of patches for all security holes detailed in "HACKMAN" [Shipley88, Shipley89] and the "Aburt archive" (two compilations detailing known computer security holes) could be considered a "milestone".

### 3.3. Installing Patches on LLNL Systems

The RAPID software system will quickly install security and bugfix patches on unclassified systems throughout LLNL. RAPID will run a system based on Track and Kerberos on a number of protected machines. These machines will be kept physically safe, and will support only patch and authentication requests (and even then, only from other LLNL machines); no other services (such as mail, login, finger, status, etc.) will be supported or provided by these machines, to eliminate any vulnerability to attack via those services.

This task will reach a milestone when all RAPID systems are being automatically monitored and patched by RAPID. A second milestone will be the first extension of RAPID service to a non-RAPID system. A third milestone will be the ability to extend RAPID service to all qualified volunteer systems.

### 3.4. Interacting With Cooperating Groups

As problems are identified and solved, RAPID will contact various other groups who also work on such problems, including the national DoE computer security center at LANL, NCSC in Maryland, CERT at Carnegie-Mellon, CSRG at Berkeley, and appropriate hardware and software vendors. By doing this in a timely manner, RAPID will avoid duplication of the effort of other groups, and will help both LLNL and others to minimize their vulnerabilities. It is expected that the RAPID center at LLNL will be a model for DoE and others to replicate; for this reason, RAPID will publish a number of technical papers describing the both the RAPID system and LLNL's experiences in implementing and operating it.

The major milestone for this task will be the identification of all significant computer security emergency response teams and other groups with RAPID-like charters, and the establishment of formal or informal cooperation agreements with each of these groups.

### 3.5. Mapping LLNL's Computer Network

Mapping will be done on a continuous basis. Automated software, based in part on systems developed by Seth Abrahams of LLNL and enhanced as appropriate by RAPID, will continually scan LLNL's network for alterations that RAPID and LLNL administrators should be aware of. These alterations include:

1. The appearance of a new, unregistered machine on the network
2. A change in name, IP address, or Ethernet address by an existing machine
3. The disappearance of a machine for an unusually long period of time
4. The presence of a machine that has not received or installed a given security patch

The mapping task will also assure that RAPID has a contact person and other standard information for each machine on the network, and that each machine is either regularly receiving

RAPID updates and patches or has has a competent system administrator who is willing and able to assure the security of the machine in some other fashion.

The first milestone of the mapping task will be the establishment of a basic map of LLNL's unclassified network. A second milestone will be the completion of the database of contact person and other information for each of the sites in the map. A third milestone will be the the completion of the software to automatically maintain and update the map information.

## 3.6. Providing a Secure Authentication Mechanism

To enable RAPID to safely install patches on all LLNL systems, and to prevent new vulnerabilities from being inadvertently introduced, a reliable, efficient, and easy-to-use secure authentication mechanism is necessary. RAPID will install Kerberos for network authentication, and will provide "shadow password" facilities (less secure than Kerberos, but more secure than existing facilities) for those applications where Kerberos is inappropriate.

Installation and use of Kerberos on all RAPID systems will be the first milestone for this task. Making Kerberos available to all other LLNL machines will be the second milestone.

## 3.7. Providing a Reliable Communication Resource

Since it is vital to maintain communications with affected and responding parties during an emergency, RAPID will provide a reliable electronic mail facility. Dynamic configuration management will allow RAPID's development machines to be substituted for each other as necessary in order to guarantee that the RAPID team, the emergency response group, and LLNL management always have a reliable communications channel.

The first and only milestone for this task will be the establishment of a communications system that meets the stated requirements.

## 3.8. Producing a RAPID Plan for VMS Systems

Currently, RAPID is primarily intended for UNIX systems. A plan will be developed to provide RAPID or RAPID-like services to VMS systems as well.

The milestone for this task will be the production of a RAPID plan for VMS.

## 3.9. Producing a RAPID Plan for Classified Systems

For security reasons, RAPID will initially only automatically install patches on unclassified systems. Patches will be made available to personnel responsible for manual installation on classified machines. A plan will be prepared for extending some or all automatic RAPID services to classified machines in manner consistent with the security requirements of those machines.

The milestone for this task will be the development of a RAPID plan for classified systems.

## 4. Requirements

The RAPID team will be a small group of programmers who will control a number of computer systems. Some of these systems will be dedicated to particular tasks, while others will be used to create patches for particular hardware and operating system combinations.

## 4.1. Personnel

The RAPID group will begin with four expert programmers and one secretary. Since the bulk of RAPID's work is programming, any member of its technical staff should be a highly competent programmer. The programming team will be composed of a system architect (who will also server as team leader, and report directly to LLNL's Computer Security Manager and the head of the Livermore Computer Center), a general UNIX and network programmer, a UNIX programmer with Kerberos and Track experience, and a UNIX programmer with software librarian and system administration experience. RAPID also expects to make use of the considerable expertise of LLNL's system programmers and administrators by working with several of them on a part-time basis (perhaps one-half day per week) in their particular areas of expertise.

After the RAPID software system is established and stabilized, a system architect will no longer be necessary, and team leadership will be passed to one of the programming experts; the system architect might then be replaced by a VMS expert (strategies for dealing with VMS are still being considered). Also, once the system is established, the Track/Kerberos programmer might be replaced with a more general UNIX expert to create system patches.

## 4.2. Equipment and Facilities

The RAPID project will require four high-power workstations and a file and compute server for system and patch development, a Macintosh II secretarial workstation, a Macintosh II with a special hardware coprocessor for semi-numeric analysis, two PostScript driven laser printers, one high-power workstation from each architecture to be supported by RAPID (including Sun 3, Sun 4, Sun 386i, and MicroVAX) for patch development, testing, and verification, three minimally-configured dedicated Sun 3/60 or MicroVAX II workstations to be Kerberos authentication servers, three medium-power file servers for patch distribution, a minimally-configured dedicated workstation for network mapping, and a MicroVAX III for network monitoring, along with various printers, modems, network connections, and other equipment. A detailed equipment list is presented in *RAPID Equipment Requirements* [RAPID89].

The project will need a machine room and six offices in close proximity to each other; five of the offices will be used for the staff and secretary, and one will be set up for "visiting experts" that will assist us from time to time and will also serve as a conference room and library. RAPID will also require telephones, a photocopier, and other standard office equipment.

## 5. Summary

The RAPID system will allow LLNL to quickly develop and install security and bugfix patches for the lab's various unclassified computer systems. It will provide LLNL with a secure authentication service. Through its mapping and monitoring functions, it will provide LLNL with a continually-updated understanding of what is *in fact* happening among the lab's unclassified computer systems. It will enhance security not only by reducting the vulnerability to attack, but by allowing LLNL to respond more quickly and effectively to any incident that *does* occur. Both RAPID and the patches it develops will be easily transportable to other parts of DoE.

**References (Partial List)**

Baldwin87

R. Baldwin, *Rule Based Analysis of Computer Security*, LCS Technical Report 401, Massachusetts Institute of Technology, 1987.

Brand87

R. Brand, "A Computer Security Tutorial", *IEEE CompCon, San Francisco 1987*, IEEE Computer Society, 1987.

Brand88

R. Brand, "A Supercomputer Security Tutorial", *IEEE Supercomputer Conference, Orlando 1988*, IEEE Computer Society, 1988.

Eichin88

M. W. Eichin and J. Rochlis, *With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988*, Massachusetts Institute of Technology, Project Athena, 29 November 1988.

Harrison88

H. E. Harrison, S. P. Shaefer, and T. S. Yoo, "Rtools: Tools for Software Management in a Distributed Computing Environment", *USENIX Conference Proceedings, Summer 1988*, pp. 85-93, USENIX Association, 20-24 June 1988.

Lord88

T. Lord, "Tools and Policies for the Hierarchical Management of Source Code Development", *USENIX Conference Proceedings, Summer 1988*, pp. 95-106, USENIX Association, 20-24 June 1988.

Michael88

G. Michael and R. Brand, "Security Considerations for Mass Storage Systems", *IEEE Mass Storage Systems Symposia, Monterey 1988*, p. 55-57, IEEE Computer Society, 1988.

Miller87

S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Slatzer, "Kerberos Authentication and Authorization System", *Project Athena Technical Plan*, Project Athena, Massachusetts Institute of Technology, 21 December 1987.

Nachbar86

D. Nachbar, "When Network File Systems Aren't Enough: Automatic Software Distribution Revisited", *USENIX Conference Proceedings, Summer 1986*, pp. 159-171, USENIX Association, June 1986.

Neuman88

C. Neuman and J. Steiner, "Authentication of Unknown Entities on an Insecure Network of Untrusted Workstations", *USENIX UNIX Security Workshop Proceedings, August 1988*, pp. 10-11, USENIX Association, 29-30 August 1988.

NSANCSC88

National Security Agency/National Center for Computer Security, *Recommendations for*

*Emergency Prevention and Recovery*, InterNet Virus Postmortem Conference, November 1988.

RAPID89
RAPID Project, *RAPID Equipment Requirements*, Lawrence Livermore National Laboratory, work in progress.

Reid87
B. Reid, "Reflections on Some Recent Widespread Computer Breakins", *Communications of the ACM*, vol. 30, no. 2, pp. 103-105, Association for Computing Machinery, February 1987.

Shipley88
P. Shipley and R. Brand, "HACKMAN -- A Systematic Study of Real Computer Security Holes", *USENIX UNIX Security Workshop Proceedings, August 1988*, USENIX Association, 29-30 August 1988.

Shipley89
P. Shipley and R. Brand, *HACKMAN: A Systematic Case Study of Computer Security*, book in progress.

Spafford88
E. H. Spafford, *The Internet Worm Program: An Analysis*, Purdue Technical Report CSD-TR-823, Purdue University, Department of Computer Sciences, 28 November 1988.

Steiner88
J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems", *USENIX Conference Proceedings, Winter 1988*, pp. 191-202, USENIX Association, February 1988.

Stoll88
C. Stoll, "Stalking the Wiley Hacker", *Communications of the ACM*, vol. 31, no. 5, pp. 484-497, Association for Computing Machinery, May 1988.

Truscott86
T. Truscott, B. Warren, and K. Moat, "A State-wide UNIX Distributed Computing System", *USENIX Conference Proceedings, Summer 1986*, pp. 499-513, USENIX Association, June 1986.

**Acknowledgements**

This paper has gone through a peer review process involving over 150 people. The authors would like to thank the following people in particular: S. Abrahams, C. Cole, D. Mansur, A. De Alvare, C. Athey, J. Carlson, G. Pavel, G. Michael, M. Medin, P. Lapsley, P. Yee, S. Grossman, K. Oberman, J. Markoff, J. Rochlis, J. Uhley, V. Subramaniam, and D. Muir.

# Configuration Management of an Environment

Susan A. Dart.
Peter Feiler.
Abstract for Usenix Software Management Workshop,
April 3-4, 1989.

Configuration management is a controlling and organizing discipline for configurations of systems. It is an important aspect of a software development environment whether it be an user's application tool or whether it be for controlling the tools within an environment. We concentrate on this latter aspect - tool maintenance within an heterogeneous environment. This problem domain encompasses tool installation, composition, integration, tailoring and upgrading. It involves managing the collection of tools available in the environment by maintaining multiple versions of tools and their configurations as well as maintaining tools that are made up of several parts (possibly of other tools) where each part is under version control. Our project deals with managing third-party tools without access to their source code.

An heterogeneous environment contains many third-party vendor tools. For instance, a typical environment may contain a public-domain mailer, window management system and editor, and a document preparer from one vendor and a design tool from another, etc. It is required to have these tools be co-resident in the same environment and integrated, that is, working in a way that does not conflict with each other. Also, it is required that the evolution of the tools such as upgrading tools and adding tools be permitted.

In an heterogeneous environment, installers need a sense of confidence that a tool can be installed and integrated with other tools in a safe manner without any deleterious side-effects. Users of tools want a sense of confidence that they can build an application making use of a consistent set of tools and have the environment maintain the history of the status of a working context. Users want to personalize a tool by tailoring it and be able to incorporate those tailorings safely into an upgraded version of the tool.

There is little automated support in an organization for configurations of tools. Tool "wizards" are the only ones who know how to install or de-install, upgrade or tailor tools. Most of the information about the tool is outside of the environment into which it needs to exist and be accessed. The user of the environment has no recourse to information about tools in the environment except for perusing the source code (if it exists) or some other documentation or contacting the wizard. By applying configuration management techniques that have been incorporated in various environments, users and installers of tools can have easier and safer access to versions of tools.

We are developing a data model for characterising third-party tools in Unix. We are implementing a prototype of our model and the configuration management techniques by using the program transformation system, Refine.

# Configuration Management of an Environment

Susan A. Dart
Peter Feiler
Abstract for Usenix Software Management Workshop
April 3-4, 1989

Configuration management is controlling and organizing the evolution of configurations of systems. It is an important aspect of a software development environment whether it is for an application tool or whether it be for controlling the tools within an environment. We concentrate on the latter aspect – tool maintenance within a heterogeneous environment. This encompasses tool installation, configuration, integration, tailoring and upgrading. It involves managing the collection of tools available in the environment by maintaining multiple versions of tools and their configurations, as well as maintaining tools that are made up of several parts (consisting of other tools) where each part is under version control. Our project deals with managing third-party tools without access to their source code.

A heterogeneous environment contains many third-party vendor tools. For instance, a typical environment may contain a public-domain mailer, a window manager, an [illegible] system and editor, and a document processor from one vendor and a graphics tool from another vendor. It is desirable to have these tools co-exist within the same environment and integrated, that is, working in a way that does not conflict with each other. Also, it is required that the evolution of the tools (e.g. upgrading tools and adding tools) be controlled.

In a heterogeneous environment, installers need a sense of confidence that a tool can be installed and integrated with other tools in a safe manner without any deleterious side-effects. Users of tools want a sense of confidence that they can build on tools making use of a consistent set of tools and have the environment maintain the history of the tools. [illegible] want to permit users a tool by tailoring [illegible] and be able to incorporate these tailorings safely into new upgraded versions of the tool.

There is little automated support in an organization for configuration of tools. What happens is that only one or two people know how to install or de-install, upgrade or tailor tools. Most of the information about the tool is [illegible] of the environment [illegible] and be accessed. The user of the environment has no access to information about tools in the environment except for reading the source code (if it exists) or some other documentation, or contacting the experts. By applying configuration management techniques that have been incorporated in various environments, users and installers of tools can have easier and safer access to versions of tools.

We are developing a data model for maintaining third-party tools in Unix. We are implementing a prototype of our model and the configuration management techniques by using the program transformation system Refine.

# CCSLAND

*Nathaniel R. Bronson III*
*bronson@Multiflow.COM*
*tan@Microvation.COM*

Multiflow Computer
Branford, Ct 06443

## 1. Introduction

CCSLAND[1] is a toolkit for the configuration management of large software systems in a UNIX environment. CCSLAND can produce multiple configurations from a single set of source files, and can reproduce multiple versions of a software system. CCSLAND extends *RCS*[2] and *make*[3], which can manage a single release of a module on a single host, to manage many releases of a large system with many modules over several hosts.

Program developers interact with CCSLAND through a set of tools which allow full freedom to develop and test modules and then check in finished modules to CCSLAND. All development is done in a local directory, and files are checked into the 'CCSLAND tree' after they have been tested.

Usually the CCSLAND system administrator serves as the software project librarian. The CCSLAND tools allow the librarian and the system architect to structure the system software to match the system architecture. These also ensure that any change to a file in the system will cause other files dependent on the change to be rebuilt.

CCSLAND supports configuration control. As a software release moves into standard manufacturing, CCSLAND provides controlled procedures for supporting and maintaining an old release concurrent with new development in subsequent releases.

## 2. Why do we need something?

The preparation of software for release varies among different development groups, leading to variablity in the quality of those releases. A software release should include every file which is part of the final tree, including all binary files, and all reproducible files should be rebuilt from their source files. The tools and libraries which are used to make a release should be saved to ensure that one can fully reproduce any object files or executables.

In some cases, software products have been released only to find out that the master source tree was missing files or has incorrect files. An example of a (poor) release process could involve placing the 'best' software into a global directory, only to result in a tree of objects not guaranteed to be reproducible from the sources in that tree. There is usually little consistency in *Makefiles*, and unless standards exists at the beginning of the project, it is difficult later to add new rules to the existing *Makefiles*.

There is a lack of Unix tools to help build large systems. While *make* and *RCS* are powerful tools, they do not deal well with multiple configurations or support multiple versions from the same source.

## 3. What does CCSLAND do for you?

CCSLAND is based on the assumption that a large percentage of the build needs of different projects are similar. Rather than having each group produce their own tools, development groups can share a single set of flexible tools. Standard tools allow the efforts of one group to shared by others.

An essential part of a software release is ensuring that the software was built with the correct compiler, include files, libraries, etc. Because releases are rarely ahead of schedule, the process of coordinating this information must be done during the development of the product, not at the last minute!

CCSLAND automates several clerical chores of a software librarian and provides the following services:

- Easily supports multiple configurations from a single set of sources.
- Checks out and checks in files with interlocks.
- Generates *Makefiles* with full dependencies.
- Controls exporting tasks, libraries, shell scripts, and header files.
- Organizes files into modules and subsystems.
- Automates subsystem and system builds.
- Controls system documents as well as executable modules.
- Creates all files in final distribution automatically.

with the following features:

- Handles binary files.
- Supports versions for the users and the librarian.
- Is layered on top of *RCS* and *make*.
- Provides full *linting* between libraries and tasks.
- Allows system builds to be done via *cron*.
- All tools are 'data' driven.
- Can use separate processors for each configuration.
- Can use *SCCS*[4] files in a limited way.

One of the powerful functions of CCSLAND is to derive all *Makefiles* from a higher level description. This ensures that all *Makefiles* have a consistent set of rules (like *clean*, *rmtasks*, etc).

## 4. CCSLAND Glossary

The following describes the keywords of CCSLAND.

| | |
|---|---|
| *system* | The base directory path where the rest of the tree lives. All CCSLAND tools need to have this specified. |
| *release* | Each *system* can optionally have multiple *releases*. Each release contains a separate set of archive (*RCS* or *SCCS*) files. |
| *checkpoint* | A *checkpoint* is a snapshot of the files in the forward process of releasing a software *system*. A *checkpoint* is a copy of the sources and one or more *configurations*. A single set of sources is used to build multiple |

| | |
|---|---|
| | configurations. |
| *configuration* | A *configuration* is a set of object files for a specific target machine (i.e. *xenix* or *sun*). CCSLAND uses a separate tree for each *configuration*. Each tree includes: all object files, any intermediate files such as libraries, and a *root* filesystem of the releasable files. |
| *chunk* | A *chunk* is a directory which reflects the partitioning of the final software products. (For an example in the BSD4.3 tree: lib/libc, bin/awk, etc). |

## 5. CCSLAND Directory Structure

For each *chunk* a directory tree exists in at least three places (using shell variables to describe the tree):

1. *$system/$release/arc/$chunk* holds the *RCS* or *SCCS* files.
2. *$system/$release/$checkpoint/src/$chunk* contains the source files which correspond to the *checkpoint*.
3. *$system/$release/$checkpoint/$configuration/obj/$chunk* contains the object files corresponding to the *configuration* (one tree for each *configuration*).

The official copy of the sources and objects are kept in a 'CCSLAND tree'. This tree is modified only by CCSLAND tools. This is where the 'official' build is done by the system librarian.

## 6. User's View of a CCSLAND System

The user accesses files from the 'CCSLAND tree' by creating a *.CCS* file which describes the *system*, and optionally the *release* and *checkpoint* The user creates a subdirectory relative to the *.CCS* file which corresponds to the *chunk*. For example with a *.CCS* file in /u/tan and CCSLAND tools which are run in the directory */u/tan/etc/fsck* will act on the *chunk /etc/fsck*.

The user accesses the 'CCSLAND tree' using the following tools:

| | |
|---|---|
| *cki* | Check files in. |
| *cko* | Check files out. |
| *ckdiff* | Compare versions of a file. |
| *cklock* | Set or break locks in a file. |
| *ckclone* | Create symbolic links to files in 'CCSLAND tree'. |
| *cklog* | Show the history for a file. |
| *makenv* | Generate a *Makefile* from a *Makemakefile*. |

The ck* tools call the corresponding *RCS* command with an appropriate pathname to the *RCS* file in the 'CCSLAND tree', and *makenv* generates a *Makefile*.

A typical session to make changes in the *chunk /etc/fsck* would involve:

1. Create a *.CCS* file.
2. Run '*ckclone etc/fsck*' to create a working directory *etc/fsck*, with symbolic links to all the corresponding files in the 'CCSLAND tree'.
3. Move into the newly-created directory *etc/fsck*.
4. Check out any files 'locked' you want to edit (*cko -l files*).
5. Run *makenv* to generate a *Makefile*.

6. Do a normal debug and editing session.
7. When satisified with the changes check them in (*cki files*).

## 7. Makemakefile Basics

The *Makemakefile* allows one to describe the targets to be built without specifying how they are to be built. *Makenv* knows to convert *.y*, *.l*, *.c*, *.s*, and *.C* files into object files and convert groups of these source files into: *programs*, *ofiles* (built using ld -rX), *libraries*, and *biglibraries* (a library built from other libraries). *Makenv* produces a *Makefile* with rules for the targets specified in the *Makemakefile*, full header dependency rules, rules to *lint* files and produce *lint libraries* as appropriate, and many psuedo targets like: *clean*, *rmlibs*, *rmtasks*, *ctags*, *cflow*, *xref*, and *listing*.

The simplest *Makemakefile* to build 'echo' and install it as */bin/echo* looks like:

```
INCLUDES=/usr/include
%incdir $(INCLUDES)
echo: { /bin/echo }
   echo.c
```

The *INCLUDES=* defines a *make* style macro and the *%incdir* defines a directory where include files can be found (*Makenv* will only search for include files in directories explicitly defined).

An example of a more advanced *Makemakefile* is one that builds two tasks, one of which should be installed 'setuid' root with a hard link. Both tasks use a local library which is built with an additional file for the *xenix* configuration. This could be described with the following Makemakefile:

```
task1: { /bin/task1 4755 root wheel /bin/linkname }
   task1.y
   tasklib.a
task2: { /bin/task2 }
   task2.l
   tasklib.a
tasklib.a:
#ifdef xenix
   xenix.c
#endif
   tasklib.c
```

A final example of a *Makemakefile* uses *%SubConfig* to generate three libraries from the same set of source files, but with different compilation flags. The file 'emulate.c' is only part of the *subconfiguration* 'emul'.

```
#
# build three different libraries from the sources in this directory
#
%SubConfig prod: -DPROD
%SubConfig debug: -DDEBUG
%SunConfig emul: -DEMULATE

#ifdef PROD
libprod.a: { /usr/lib/libprod.a }
#endif
#ifdef DEBUG
libdebug.a: { /usr/lib/libdebug.a }
#endif
```

```
#ifdef EMUL
libemul.a: { /usr/lib/libemul.a }
   emulate.c          # need this extra file for emulation
#endif
   file.y
   file2.l
   file3.c
```

## 8. Other Makemakefile features

There are many options to control what is produced in the *Makefile*. All generated *Makefiles* include a *Make template* file, which can conditionally define macros (like CC) using *cpp*-like ifdefs.

This is a sample of the *Makemakefile* directives.

| | |
|---|---|
| *%xstr* | Compile targets using **xstr**(1). |
| *%noexportlibs* | By default libraries are exported to the 'root', but they can be kept in a separate tree if they are not meant to be distributed as part of the released product. |
| *%gnumake* | Produce a *Makefile* using the GNU *make*(1).[5] |
| *%alttskld* | Build a task with an alternate loader command (can be used to build with different libraries). |
| *%native* | Build a task to run in the local environment. This can be necessary when cross-compiling. |
| *%nmake*[6] | This flags currently only converts comments to that required by *nmake* and disables the header dependency generation. *Makenv* generates large *Makefiles*, so the compilation feature of *nmake* might be useful. |
| *%ExportMan* | Export a manual page (into a subdirectory using the extension). |
| *%makedepends* | All objects depend on the *Makefile*. |
| *%depends* | Calculate full dependencies for all source files. |
| *%define* | *Makenv* must know of all defined symbols so it can process the *Makemakefile* and calculate header dependencies. |

## 9. System Management Tools

The basic system management tools are:

| | |
|---|---|
| *buildsystem* | The main build program (described below). |
| *emaster* | Edit the 'database' kept in a 'Master' directory under the *release*). |
| *sharemaster* | Distribute the 'Master' files to each checkpoint. |
| *markcheckpoint* | Mark all files part of the current checkpoint with the checkpoint name. This creates an *RCS* branch in each file. |
| *buildcheckpoint* | Build one or more checkpoints (calls buildsystem, sharemaster and markcheckpoint). |
| *fixcofiles* | Verify all checked out files match their *RCS* files. |

There are several directories used by the system administration tools:

1. *$system/$release/Master*: This holds a copy of the 'database' files, which are then distributed to the 'misc' directory below.
2. *$system/$release/$checkpoint/misc*: This directory contains the 'database' files which are unique to this *checkpoint*. Once *markcheckpoint* has been run these files are treated as read-only.

## 10. The Buildsystem Program

*Buildsystem* applies the *action* specified to each *chunk*. By default the *actions* are:

- Verify the directory tree exists.
- Determine what *chunks* have what *configurations*.
- Build *Makefiles*.
- Make any directories and symbolic links needed in the 'root' for each *configuration*.
- Build and install headers.
- Build and install libraries.
- Build and install big libraries.
- Build and install tasks.
- Build and install any tasks which require previous tasks to exist. (*finalphase*)
- Build and install shell scripts.
- Install manual pages.
- Install source files.
- Check for duplicate files being installed from different *directories*.
- Generate a bill of materials for each *configuration*.

After each phase is complete any chunk which had errors building *Makefiles* (for example because a library was missing) will have their *Makefile* rebuilt. *Makenv* and *buildsystem* communicate to avoid the overhead of invoking *make* if there are no targets of a given type.

*Buildsystem* assumes that any of the above *action*, except *finalphase*, can be done to parallel to all *chunks*, except that all *chunks* must have *action* applied to them before the next *action* can be started on any *chunk*. For example libraries can be built in parallel, but they all must be finished before tasks can be built.

## 11. Future Directions

Many of the 'database' files were appropriate when CCSLAND was a collection of shell scripts, but now that the build tools are written in C, the large number of one line files should be reduced.

The ability to specify regressions tests to run before certain files are installed will be added.

The ck* tools should run setgid.

*Makenv* should be completely table-driven. The current version knows about C, yacc, lex, assembly language, and lint. C++ is not completely supported yet and the current implementation involves changing C code.

## 12. Why not use something else?

When the CCSLAND project was started in 1985 NSE[7] did not exist. When NSE became available, it looked promising. It provides a virtual filesystem on a per release basis, but on closer inspection it mainly replaced *RCS* or *SCCS* and did not include any tools to help the build process. NSE has a few other shortcomings:

- It can not accept *RCS* files. (needs a rcstosccs)
- It hides the directory structure from you.
- It requires kernel modifications to work.
- It was slow on our 3/50s.
- It requires expensive licensing to run on non-Sun systems.

Another alternative briefly explored was SoftTools CCC; however the expense and complexity of CCC were deterents. Furthermore, having all source code in a foreign database could hinder recovery from disk corruption problems.

The SPMS[8] Software Project Management System has several notable features:

- *Priority* is used to determine the order in which an action is applied to a list of directories.
- *Types* are used to tag all directories of a certain type. These tags can be used by the command *pexec* to execute a command on all directories of the specified *type*.
- The directory structure is hidden from the user, and to access the tree a series of SPMS commands must be used.
- A mechanism for testing is built-in.
- SPMS supports a single tree. Multiple versions can not be concurrently.

CCSLANDs *makenv* can handle complex source directory trees, and the *Makemakefile* can be conditional per configuration. SPMS has *mkmf*, but it works best for one target type (library or program) per directory and has no conditional capability. NSE requires user provided *Makefiles*.

No software product is as important to the survival of a company as the software which maintains their source files. The most minor bug can waste precious time, so any solution must have the full source available. CCSLAND is provided with source, which allows users to extend it to their needs.

## 13. Summary

CCSLAND combines the power of *make*, *RCS*, symbolic links, and the Unix filesystem to provide a system which is flexible enough to support most needs. *make* and *RCS* individually provide the foundation for dealing with a directory at a time, while CCSLAND allows one to have a complex directory structure like that needed to produce a large software product.

CCSLAND was designed to be simple enough for people to understand how it works and to be expanded as better ideas come along. CCSLAND is designed to be used by people (like myself) who already have code which exists in *SCCS* or *RCS* files and who do not have the resources to convert to a fancy CASE product. CCSLAND has its share of complexity, but building large software systems is not always simple. CCSLAND is in use to build system software ranging from: Unix kernels, the Unix user state, a large amount of cross-compiled code, and code which runs in EPROM. Most of the features of CCSLAND were to fix a very specific problem. By making the solution general-purpose, the majority of these 'fixes' have been useful for other products.

CCSLAND has evolved for more than four years. It started as a series of shell scripts and was used to release software which ran on a Vax and a small 68k Unix system. A different product group had the *Makefile* generator on which the current CCSLAND version is based. Release 1.0 integrated the above tools, included tools to make the directory structure (which was previously built manually), and supported multiple versions for each *chunk*. Release 2.0 was mostly the conversion of the user tools to C (which sped things up by a factor of 10). Release 4.2 incorporated a new directory structure in order to facilitate allocation; this was a problem is previous versions because all configurations were under the same tree. In addition, file cloning was added and the build tools were rewritten in C. The later feature has significantly reduced the overhead of doing a build.

I would like to thank all the people and patient users who have helped make CCSLAND successful. John Nelson, Gordon Weekly, Rick Frerichs, Don Bolinger, and Chris Ryland deserve particular mention.

## *REFERENCES*

1. N. R. Bronson III, "CCSLAND Release 2.4 - User Manual"
2. W. F. Tichey, "An Introduction to the Revision Control System", Programmer Supplement Documents, PS1:13 of BSD4.3 Manuals.
3. S. I. Feldman, "Make - A Program for Maintaining Computer Programs", Programmer Supplement Documents, PS1:12 of BSD4.3 Manuals.
4. E. Allman,"An Introduction to the Source Code Control System", Programmer Supplement Documents, PS1:14 of BSD4.3 Manuals.
5. R. M. Stallman, R. McGrath "GNU make - A Program for Directing Compilation"
6. G. S. Fowler, "The Fourth Generation Make"
7. "Network Software Environment: Reference Manual", Sun Microsystems
8. P. J. Nicklin, "The SPMS Software Project Management System"

# The Release Engineering of 4.3BSD

*Marshall Kirk McKusick*

*Michael J. Karels*

*Keith Bostic*

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

*ABSTRACT*

This paper describes an approach used by a small group of people to develop and integrate a large software system. It details the development and release engineering strategy used during the preparation of the 4.3BSD version of the UNIX† operating system. Each release cycle is divided into an initial development phase followed by a release engineering phase. The release engineering of the distribution is done in three steps. The first step has an informal control policy for tracking modifications; it results in an alpha distribution. The second step has more rigid change mechanisms in place; it results in a beta release. During the final step changes are tracked very closely; the result is the final distribution.

## 1. Introduction

The Computer Systems Research Group (CSRG) has always been a small group of software developers. This resource limitation requires careful software-engineering management as well as careful coordination of both CSRG personnel and the members of the general community who contribute to the development of the system.

Releases from Berkeley alternate between those that introduce major new facilities and those that provide bug fixes and efficiency improvements. This alternation allows timely releases, while providing for refinement, tuning, and correction of the new facilities. The timely followup of "cleanup" releases reflects the importance CSRG places on providing a reliable and robust system on which its user community can depend.

The development of the Berkeley Software Distribution (BSD) illustrates an *advantage* of having a few principal developers: the developers all understand the entire system thoroughly enough to be able to coordinate their own work with that of other people to produce a coherent final system. Companies with large development organizations find this result difficult to duplicate. This paper describes the process by which the development effort for 4.3BSD was managed [Leffler *et al.*, 1989].

†UNIX is a registered trademark of AT&T in the US and other countries.

## 2. System Development

The first phase of each Berkeley system is its development. CSRG maintains a continuously evolving list of projects that are candidates for integration into the system. Some of these are prompted by emerging ideas from the research world, such as the availability of a new technology, while other additions are suggested by the commercial world, such as the introduction of new standards like POSIX, and still other projects are emergency responses to situations like the Internet Worm.

These projects are ordered based on the perceived benefit of the project as opposed to its difficulty; the most important are selected for inclusion in each new release. Often there is a prototype available from a group outside CSRG. Because of the limited staff at CSRG, this prototype is obtained to use as a starting base for integration into the BSD system. Only if no prototype is available is the project begun in-house. In either case, the design of the facility is forced to conform to the CSRG style.

Unlike other development groups, the staff of CSRG specializes by projects rather than by particular parts of the system; a staff person will be responsible for all aspects of a project. This responsibility starts at the associated kernel device drivers; it proceeds up through the rest of the kernel, through the C library and system utility programs, ending at the user application layer. This staff person is also responsible for related documentation, including manual pages. Many projects proceed in parallel, interacting with other projects as their paths cross.

All source code, documentation, and auxiliary files are kept under a source code control system. During development, this control system is critical for notifying people when they are colliding with other ongoing projects. Even more important, however, is the audit trail maintained by the control system that is critical to the release engineering phase of the project described in the next section.

Much of the development of BSD is done by personnel that are located at other institutions. Many of these people not only have interim copies of the release running on their own machines, but also have user accounts on the main development machine at Berkeley. Such users are commonly found logged in at Berkeley over the Internet, or sometimes via telephone dialup, from places as far away as Massachusetts or Maryland, as well as from closer places, such as Stanford. For the 4.3BSD release, certain users had permission to modify the master copy of the system source directly. People given access to the master sources are carefully screened beforehand, but are not closely supervised. Their work is checked at the end of the beta-test period by CSRG personnel who back out inappropriate changes. Several facilities, including the Fortran and C compilers, as well as important system programs, for example, **telnet** and **ftp**, include significant contributions from people who did not work directly for CSRG. One important exception to this approach is that changes to the kernel are made only by CSRG personnel, although the changes are often suggested by the larger community.

The development phase continues until CSRG decides that it is appropriate to make a release. The decision to halt development and transition to release mode is driven by several factors. The most important is that enough projects have been completed to make the system significantly superior to the previously released version of the system. For example, 4.3BSD was released primarily because of the need for the improved networking capabilities and the markedly improved system performance. Of secondary importance is the issue of timing. If the releases are too infrequent, then CSRG will be inundated with requests for interim releases. Conversely, if systems are released too frequently, the integration cost for many vendors will be too high, causing them to ignore the releases. Finally, the process of release engineering is long and tedious. Frequent releases slow the rate of development and cause undue tedium to the staff.

## 3. System Release

Once the decision has been made to halt development and begin release engineering, all currently unfinished projects are evaluated. This evaluation involves computing the time required to complete the project as opposed to how important the project is to the upcoming release. Projects that are not selected for completion are removed from the distribution branch of the source code control system and saved on branch deltas so they can be retrieved, completed, and merged into a future release; the remaining unfinished projects are brought to orderly completion.

Developments from CSRG are released in three steps: alpha, beta, and final. Alpha and beta releases are not true distributions—they are test systems. Alpha releases are normally available to only a few sites, usually those working closely with CSRG. More sites are given beta releases, as the system is closer to completion, and needs wider testing to find more obscure problems. For example, 4.3BSD alpha was distributed to about fifteen sites, while 4.3BSD beta ran at more than a hundred.

### 3.1. Alpha Distribution Development

The first step in creating an alpha distribution is to evaluate the existing state of the system and to decide what software should be included in the release. This decision process includes not only deciding what software should be added, but also what obsolete software ought to be retired from the distribution. The new software includes the successful projects that have been completed at CSRG and elsewhere, as well as some portion of the vast quantity of contributed software that has been offered during the development period.

Once an initial list has been created, a prototype filesystem corresponding to the distribution is constructed, typically named **/nbsd**. This prototype will eventually turn into the master source tree for the final distribution. During the period that the alpha distribution is being created, **/nbsd** is mounted read-write, and is highly fluid. Programs are created and deleted, old versions of programs are completely replaced, and the correspondence between the sources and binaries is only loosely tracked. People outside CSRG who are helping with the distribution are free to change their parts of the distribution at will.

During this period the newly forming distribution is checked for interoperability. For example, in 4.3BSD the output of context differences from **diff** was changed to merge overlapping sections. Unfortunately, this change broke the **patch** program which could no longer interpret the output of **diff**. Since the change to **diff** and the **patch** program had originated outside Berkeley, CSRG had to coordinate the efforts of the respective authors to make the programs work together harmoniously.

Once the sources have stabilized, an attempt is made to compile the entire source tree. Often this exposes errors caused by changed header files, or use of obsoleted C library interfaces. If the incompatibilities affect too many programs, or require excessive amounts of change in the programs that are affected, the incompatibility is backed out or some backward-compatible interface is provided. The incompatibilities that are found and left in are noted in a list that is later incorporated into the release notes. Thus, users upgrading to the new system can anticipate problems in their own software that will require change.

Once the source tree compiles completely, it is installed and becomes the running system that CSRG uses on its main development machine. Once in day-to-day use, other interoperability problems become apparent and are resolved. When all known problems have been resolved, and the system has been stable for some period of time, an alpha distribution tape is made from the contents of **/nbsd**.

The alpha distribution is sent out to a small set of test sites. These test sites are selected as having a sophisticated user population, not only capable of finding bugs, but also of determining their cause and developing a fix for the problem. These sites are usually composed of groups that are contributing software to the distribution or groups that have a particular expertise with some portion of the system.

## 3.2. Beta Distribution Development

After the alpha tape is created, the distribution filesystem is mounted read-only. Further changes are requested in a change log rather than being made directly to the distribution. The change requests are inspected and implemented by a CSRG staff person, followed by a compilation of the affected programs to ensure that they still build correctly. Once the alpha tape has been cut, changes to the distribution are no longer made by people outside CSRG.

As the alpha sites install and begin running the alpha distribution, they monitor the problems that they encounter. For minor bugs, they typically report back the bug along with a suggested fix. Since many of the alpha sites are selected from among the people working closely with CSRG, they often have accounts on, and access to, the primary CSRG development machine. Thus, they are able to directly install the fix themselves, and simply notify CSRG when they have fixed the problem. After verifying the fix, the affected files are added to the list to be updated on **/nbsd**.

The more important task of the alpha sites is to test out the new facilities that have been added to the system. The alpha sites often find major design flaws or operational shortcomings of the facilities. When such problems are found, the person in charge of that facility is responsible for resolving the problem. Occasionally this requires redesigning and reimplementing parts of the affected facility. For example, in 4.2BSD, the alpha release of the networking system did not have connection queueing. This shortcoming prevented the network from handling many connections to a single server. The result was that the networking interface had to be redesigned to provide this functionality.

The alpha sites are also responsible for ferreting out interoperability problems between different utilities. The user populations of the test sites differ from the user population at CSRG, and, as a result, the utilities are exercised in ways that differ from the ways that they are used at CSRG. These differences in usage patterns turn up problems that do not occur in our initial test environment.

The alpha sites frequently redistribute the alpha tape to several of their own alpha sites that are particularly interested in parts of the new system. These additional sites are responsible for reporting problems back to the site from which they received the distribution, not to CSRG. Often these redistribution sites are less sophisticated than the direct alpha sites, so their reports need to be filtered to avoid spurious, or site dependent, bug reports. The direct alpha sites sift through the reports to find those that are relevant, and usually verify the suggested fix if one is given, or develop a fix if none is provided. This hierarchical testing process forces bug reports, fixes, and new software to be collected, evaluated, and checked for inaccuracies by first-level sites before being forwarded to CSRG, allowing the developers at CSRG to concentrate on tracking the changes being made to the system rather than sifting through information (often voluminous) from every alpha-test site.

Once the major problems have been attended to, the focus turns to getting the documentation synchronized with the code that is being shipped. The manual pages need to be checked to be sure that they accurately reflect any changes to the programs that they describe. Usually the manual pages are kept up to date as the program they describe evolves. However, the supporting documents frequently do not get changed, and must be edited to bring them up to date. During this review, the need for other documents becomes evident. For example, it was during this phase of 4.3BSD that it was decided to add a tutorial document on how to use the socket interprocess communication primitives.

Another task during this period is to contact the people that have contributed complete software packages (such as **RCS** or **MH**) in previous releases to see if they wish to make any revisions to their software. For those who do, the new software has to be obtained, and tested to verify that it compiles and runs correctly on the system to be released. Again, this integration and testing can often be done by the contributors themselves by logging directly into the master machine.

After the stream of bug reports has slowed down to a reasonable level, CSRG begins a careful review of all the changes to the system since the previous release. The review is done by running a recursive **diff** of the entire source tree—here, of **/nbsd** with 4.2BSD. All the changes are checked to ensure that they are reasonable, and have been properly documented. The process often turns up questionable changes. When such a questionable change is found, the source code control system log is examined to find out who made the change and what their explanation was for the change. If the log does not resolve the problem, the person responsible for the change is asked for an explanation of what they were trying to accomplish. If the reason is not compelling, the change is backed out. Facilities deemed inappropriate in 4.3BSD included new options to the directory-listing command and a changed return value for the *fseek*() library routine; the changes were removed from the source before final distribution. Although this process is long and tedious, it forces the developers to obtain a coherent picture of the entire set of changes to the system. This exercise often turns up inconsistencies that would otherwise never be found.

The outcome of the comparison results in a pair of documents detailing changes to every user-level command [McKusick *et al.*, 1986] and to every kernel source file [Karels, 1986]. These documents are delivered with the final distribution. A user can look up any command by name and see immediately what has changed, and a developer can similarly look up any kernel file by name and get a summary of the changes to that file.

Having completed the review of the entire system, the preparation of the beta distribution is started. Unlike the alpha distribution, where pieces of the system may be unfinished and the documentation incomplete, the beta distribution is put together as if it were going to be the final distribution. All known problems are fixed, and any remaining development is completed. Once the beta tape has been prepared, no further changes are permitted to **/nbsd** without careful review, as spurious changes made after the system has been **diff**ed are unlikely to be caught.

### 3.3. Final Distribution Development

The beta distribution goes to more sites than the alpha distribution for three main reasons. First, as it is closer to the final release, more sites are willing to run it in a production environment without fear of catastrophic failures. Second, more commercial sites delivering BSD-derived systems are interested in getting a preview of the upcoming changes in preparation for merging them into their own systems. Finally, because the beta tape has fewer problems, it is beneficial to offer it to more sites in hopes of finding as many of the remaining problems as possible. Also, by handing the system out to less sophisticated sites, issues that would be ignored by the users of the alpha sites become apparent.

The anticipation is that the beta tape will not require extensive changes to either the programs or the documentation. Most of the work involves sifting through the reported bugs to find those that are relevant and devising the minimal reasonable set of changes to fix them. After throughly testing the fix, it is listed in the update log for **/nbsd**. One person at CSRG is responsible for doing the update of **/nbsd** and ensuring that everything affected by the change is rebuilt and tested. Thus, a change to a C library routine requires that the entire system be rebuilt.

During this period, the documentation is all printed and proofread. As minor changes are made to the manual pages and documentation, the affected pages must be reprinted.

The final step in the release process is to check the distribution tree to ensure that it is in a consistent state. This step includes verification that every file and directory on the distribution has the proper owner, group, and modes. All source files must be checked to be sure that they have appropriate copyright notices and source code control system headers. Any extraneous files must be removed. Finally, the installed binaries must be checked to ensure that they correspond exactly to the sources and libraries that are on the distribution.

This checking is a formidable task given that there are over 20,000 files on a typical distribution. Much of the checking can be done by a set of programs set to scan over the distribution tree. Unfortunately, the exception list is long, and requires hours of tedious hand checking; this has caused CSRG to develop even more comprehensive validation programs for use in our next release.

Once the final set of checks has been run, the master tape can be made, and the official distribution started. As for the staff of CSRG, we usually take a brief vacation before plunging back into a new development phase.

## References

Karels, 1986.
M. J. Karels, "Changes to the Kernel in 4.3BSD," pp. 13:1—32 in *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version,* USENIX Association, Berkeley, CA (1986).

Leffler *et al.*, 1989.
S. J. Leffler, M. K. McKusick, M. J. Karels, & J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA (1989).

McKusick *et al.*, 1986.
M. K. McKusick, J. M. Bloom, & M. J. Karels, "Bug Fixes and Changes in 4.3BSD," pp. 12:1—22 in *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, USENIX Association, Berkeley, CA (1986).

# Project Athena's Release Engineering Tricks

Don Davis, MIT Staff
E40-319, MIT, Cambridge, MA 02139

*ABSTRACT*

This paper discusses Project Athena's approach to preparing software releases. As a large, centrally-administered UNIX network, Athena has had to solve some difficult software-management problems. Athena's Release Engineering Group relies on *easily-taught* work-disciplines and procedures, and on a *few* high-leverage tools, so as to avoid cultivating the usual reliance on release-building expertise.

## Introduction

Project Athena is the largest *centrally-administered* workstation network in the world; we have 850 cpu's: 600 μVAXen, 200 IBM RT/PCs, and 50 VAX servers, spread over 23 Ethernets and a Proteon proNET-10 fiber-optic spine. We've merged features of 4.3BSD, IBM's AOS 4.3, DEC's ULTRIX 2.0, and Sun's NFS, and have added many features of our own, including the X Window System, Kerberos authentication service, Hesiod name service, Zephyr notification service, and the Moira Service Management System.[1] Almost all of our code runs without modification on both VAXen and RTs. Our additions comprise ~20M of source-code. We will soon begin to evaluate IBM 8570 (386-based PS/2) and DECstation 3100 (PMAX) workstations, and are evaluating others now. We have about 20 full-time system programmers overall.

Of Athena's workstations, 600 are installed in public areas. Almost all of our users are highly-pressured, computer-naive undergraduate students. We currently have about 11,000 users; 5,000 of them login at least once per week. Our system serves about 3,000 students each day. Multiply these numbers by the fact that most of these people never get enough sleep, and you can see that we *suffer* when the system breaks!

Central administration of such a large net is difficult; indeed, most of our in-house development has been invested in the aforementioned system administration tools. We at MIT *had* to make this investment, because our users can't be expected to help keep the system alive. Thus, our most carefully guarded resource is not the spine-bandwidth of 10 Mbit/sec, but our 5 Operations system-programmers. Consider that figure: five people, to run 850 machines! We claim to be able to expand to 10,000 workstations; in fact, we have only 850 for political reasons, not technical ones. In brief, Athena's central constraints are:

- "No shoe leather."
- "No network broadcast."
- "Does it scale well?"
- "The sponsor is always right."

[1]G.W. Treese, "Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD." In *USENIX Conference Proceedings* (February 1988), pp. 175-182.

Athena's Release Engineering staff are 3 of the 15 System Development programmers. Our priorities are:

1. providing identical software functionality on all types of μVAXen and RTs;

2. facilitating a centralized administration;

3. stabilizing the system, so that releases can be more invisible, frequent, and automatic;

4. merging the VAX and RT source trees (mostly complete);

5. smoothing the release cycle, so that a new student-hire can turn the crank.

Priority #1 is an ongoing chore, requiring a particular work-discipline. #2 was the motivation for many of Athena's added services. #4 remains an episodic chore, and we expect no relief. Priority #5, "Smoothing the Release Cycle," is the main topic of this article.

## Software Releases at Athena

An Athena release consists of two sets (VAX and RT) of 12,000 files, plus update scripts and installation media. We provide educational software separately; it's *not* in the release. We keep a minimum (4M VAX, 6.5M RT) of software on the workstations' local disks. The workstations usually get executables from Remote Virtual Disk (RVD) "binary-servers", but may also use NFS or AFS for this purpose.[2] These servers' contents and the workstations are updated around 3-4 times per year; we recompile the clean source tree about once per year. Most workstations are updated automatically, without a human visit. Each workstation initiates its own update asynchronously. The servers' own software update is a modification of the workstation update.

Athena's release-cycle is complicated by several of our system's features:

**Common User Interface**: We insist that the VAX and RT user-interfaces be as similar as keyboard-differences permit. To achieve this, we've merged 85% of the two hosts' source-trees. New releases from Berkeley, IBM, and Sun get merged anew, at great cost. At release-time, we compare the VAX and RT releases for inode-level differences (uid, gid, type, and mode).

**Local Disks**: All workstations are dataless nodes; they have local disks for swap, and for always-resident executables: kernel, network code, X server, shells, etc. We're trying to decrease these local root contents, so as to reduce the workstations' update-frequency. We'd like to be able to make some releases by updating only the binary-servers, without updating the workstations' roots, but our kernels aren't yet static enough.

**Remote Virtual Disk**: To ship binaries, we usually use Remote Virtual Disk (RVD) service, which is faster than NFS. RVD is a lower-level protocol than NFS; while NFS-requests access *files*, RVD-requests access *disk-blocks*. We've had to put a lot of work into hiding RVD's grotesque user-interface from our users, but RVD has one redeeming feature: it is very hard to

[2]All of our kernels support both RVD and NFS ( Sun Microsystem's Network File System). We're now preparing to add CMU's AFS ( Andrew File System, previously known as VICE).

write to an RVD pack from which another user is reading. This is convenient for our central administration needs, because it prevents any field-modification of binary-servers' contents. Thus, when a user complains of a bug, we don't have to check his binary-server's contents for subnet-specific "fixes;" we know that our master copy of those contents is identical to all others.

**Two-Piece Releases**: We ship releases in two filesystems: less than 10M of root, /bin, /dev, /etc, and /lib; and 110M of /usr. The local root-contents all come from the smaller pack. We try to keep all of the volatile code in the smaller pack, too, so that the larger one needs less-frequent updates. The small pack is now *too* small for RT releases, so that the 30 RVD servers' disks need to be reformatted.

**Service Interdependence**: Athena's additions to the UNIX environment ( Kerberos, X, Zephyr, Hesiod, Moira) all depend on each others' libraries. Further, several of these subsystems are in active development, so that half of the annual "clean-source build" is taken up with repeated efforts to meld these systems' modifications.

## Smoothing the Release-Cycle

To repeat: our first priority here is to reduce Release Engineering's reliance on expertise. To be blunt, experts not only can die, they do quit, and MIT doesn't pay enough to keep them all from quitting. Our goal is to "haze" all new student-hires, by having each one do a release.

### Work-Disciplines

The simplest improvement we've made in our release-cycle is to follow some easily-taught work-disciplines:

**Source Access**: Only Release-Engineering members change the central source-trees. This allows us to ensure that the VAX and RT copies of the sources remain identical.

**Specification**: Management leads in the construction of the "Release Notes" document, which specifies in advance what is and is not to go into the release. This preparation is a political free-for-all and the published document is thus invaluable as a contract; even better, Release Engineering gets to edit the final draft, once the release is ready for distribution.

**One Person/ One Build**: We divide the release into two parallel efforts: one person invokes "make clean; make all; make install" in the VAX source-tree, another does the same thing in the RT source-tree, and a third plays gofer for both. The gofer helps by shielding the builders from distractions.

**Worklogs and Makelogs**: The two builders build **make** logfiles, and keep personal work logs as well. All three parties communicate via a *private* conferencing system.[3] We have found that work logs are crucial, because they record last-minute source changes and *why* they were

[3]At Athena, we use **discuss**, an authenticated network conferencing system. See Ken Raeburn, Jon Rochlis, William Sommerfeld, and Stan Zanarotti, "*Discuss*: An Electronic Conferencing System for a Distributed Computing Environment." In *USENIX Conference Proceedings* (Winter 1989).

necessary. RCS logs are necessary, too, but are *not* sufficient, because they are distributed throughout the two 800M source-trees.

**No Compiler Warnings**: We try to make everything compile, load, and install correctly, without warning-messages. This makes it easier to detect new breakage in the makelogs, which run to thousands of lines. This goal was hard to achieve, because **hc**, the IBM C compiler for the RT, offers lots of **lint**-like warnings. To do this, we've had to tweak a lot of source, and most of our 1300 Makefiles.

**One Person/ Two Fixes**: Finally, the three workers check that each change promised by the Release Notes is in place. Here, they specialize differently: each person ensures that a given change has been installed correctly for *both* host-types, logging his changes as usual. *Only with the invention of this discipline* did Athena meet the goal of "identical software functionality on VAXen and RTs."

## Release Tools

**Prot_sources** is a tool that massages the sources' access permissions every night. This allows builders to start a build at any time, without using root access. **Prot_sources** knows a great deal about source-trees, and is tuned for speed; it traverses our VAX sources (800M) in about 1.5 hours.

The tool's purpose is two-fold: to facilitate grandiose, system-building "make all" and "make clean" invocations, and to touch up security holes. In other words, we want the sources to be protected neither too tightly, nor too loosely, but only "just right." The command-line runs:

```
prot_sources [-R] [pathname] [pathname] ...
```

Each pathname argument may be a filename or a directory-name. The default is the current directory. If **-R** is given, any directories in the pathname-list will be scanned recursively. Symbolic links to directories are NOT descended. There are 4 classes of protection:

| *Classification* | *uid.gid* | *mode* |
|---|---|---|
| WRITABLE | builder.builders | 664 |
| EXECUTABLE | builder.builders | 775 |
| LOCKED_SOURCES | *author*.builders | o-w |
| UNLOCKED_SOURCES | builder.builders | 444 |

**Prot_sources** classifies and protects the source-tree as follows:

• WRITABLE: symlinks, object-files, vers.c files in kernel source-trees, **lex**- and **yacc**-generated C sources, everything in **config**'s kernel-directories, Makefile~, core, a.out, and other special cases.

• EXECUTABLE: directories, executable files.

• LOCKED_SOURCES: *.[chflys], Makefiles, and their RCS-files, unrecognized names.

• UNLOCKED-SOURCES: *.[chflys], Makefiles, and their RCS-files.

Our use of **prot_sources** has proven very successful, in that our **make** invocations are very seldom derailed by broken permissions.

**Track** is an **rdist**-like tool for updating filesystems which was developed at BellCore.[4] We ported it to the RT, added some features of our own, and rewrote it pretty completely for speed and maintenance. Compared to **rdist**, **track** offers the following (dis)advantages:

• Net-Performance: **Track** optimizes its use of the network for one-to-many distribution, by "compiling" the exporting filesystem's inodes' contents into a single file, the *statfile*. When an updating host runs **track** for an update, **track** compares the local file system against the *statfile*'s contents. This saves thousands of networked stat() calls.

• Net-Politics: While **rdist** pushes updates at its clients, **Track** offers a "librarian-subscriber" model of file-distribution. That is, **track** allows subscribers to be selective about what they import of the librarian's exports. Also, since a subscriber updates by running **track** himself, he decides when and whether to update.

• Flexibility: **Track** allows the export and/or import of symlinks to the librarian's files and directories. **Track** can update device nodes, so that we run **MAKEDEV** not on our workstations, but on the RVD-packs' master copy of /dev. We use these features to replace a lot of error-prone update-script code. Finally, **track**'s *statfile*-output is designed to be filtered through **awk**, allowing **find**-like selectivity. **Rdist** offers none of these features.

• Bad User-Interface: **Track**'s user-interface is even worse than **rdist**'s. The command-line, *subscription-list* syntax, and overall semantics are all user-hostile. We're rewriting that code now.

We use **track** to update workstations, servers, and even source-trees. Usually, we construct one or more **track** invocations into update-scripts. We also use **awk**'ed *statfiles* to compare different releases for directory and inode-differences, where **diff** would report irrelevant file-content differences. This automated checking of 12,000 files is important, because hand-checking and staff-testing just haven't ever sufficed.

A surprisingly high-leverage trick was to add entry and exit banners to **make**, which enable junior staff to read **make**'s logfiles. We also equipped **make** to read the environment variable MAKEOPT, so that recursive invocations can be forced to print these banners. Before we added these banners, it took a lot of expertise to diagnose and fix a broken build. Remember, we're talking about 1300 Makefiles!

**Makesrv** is our script-set for converting a standard public-workstation configuration to a server's. By "configuration", here, we mean root-contents; all servers run the same kernel version. The scripts use **track**, so that their maintenance is usually just a matter of changing various *subscription-lists* that the **track** invocations require. **Makesrv** currently configures servers of the following types:

| | | | | |
|---|---|---|---|---|
| NFS | Hesiod | Kerberos | Moira | On-Line-Consulting |
| RVD | Zephyr | Discuss | Printer | |

[4]Daniel Nachbar, "When Network File Systems Aren't Enough: Automatic File Distribution Revisited." In *USENIX Conference Proceedings* (Summer 1986).

## Enduring Problems

**Service Interdependence** (mentioned above) boils down to two problems: (1) parallel development always incurs a merging cost; (2) new technology has to bootstrap itself. Now, merging and bootstrapping are inherent to the R&D lifestyle; they're not going to go away. It may help, though, to build *all* libraries before we build *any* of our programs, even at the expense of complicating some Makefiles. This would relieve the release-cycle of some interdependence-spawned delays, since we would rebuild less code in each pass through the code-merge cycle.

**RT executables** are bigger than VAX executables. This comes partly from the RTs more prolix RISC architecture, but mostly comes from printf() calls. RTs are capable of running without hardware support for floating-point, so every program that calls printf() gets linked to a gung-ho floating-point library. Thus, even though all of our RTs have hardware FPPs, and even though very few of the programs use floats, executables bulk 30% larger on the RT than on the VAX. This makes for space problems, wherever RT executables live. Either we have to reformat a lot of disks, or we have to add a dynamic-linking scheme to the RT kernel. Other solutions are possible, but they're even less attractive.

**Our developers' handoff** to Release Engineering is broken; we'd like to retain control of our central source-trees, without actually installing bug-fixes ourselves. We haven't devised a workable balance between security and laziness. Also, developers can't fully test their own Makefiles for compatibility with our build-procedures, because their development environments are different from ours. For example, developers have their own access-control groups, which are mutually exclusive with Release Engineering's "builders" group. This often leads to permissions-breakage when we try to build their code.

**Preparing installation media** still requires expertise. Currently, to install (or re-install) a workstation, we use the boot floppies' kernel and scripts to give the workstation remote file access. The workstation then runs a remotely-held script that reformats the hard disk and updates it to the current workstation configuration. The hard part is to prepare a very compact, network-capable configuration that fits on our various removable media.[5] We currently have to hand-craft a kernel and boot-blocks to fit, each time we cut a release. To solve this problem, we're working on a stand-alone **tftp** boot program, which will fit on all of the removable media. At install-time, this primary boot will download a kernel that has a memory-resident root-partition.

## Appendix: Trademarks

UNIX is a registered trademark of AT&T. The X Window System is a trademark of MIT. DECstation, PMAX, ULTRIX, VAX, and μVAX are trademarks of Digital Equipment Corporation. AOS, PS/2, RT/PC, and RVD are trademarks of International Business Machines Corporation. Sun and NFS are trademarks of Sun Microsystems. proNET-10 is a trademark of Proteon, Inc.

[5] RT-floppies, RX33's, RX50's, TK50's, and RL02's. The smallest of these allows 350K for file-contents.

# Configuration Management in the X Window System

*Jim Fulton*

X Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

*ABSTRACT*

The X Window System† has become an industry standard for network window technology in part because of the portability of the sample implementation from MIT. Although many systems are designed to reuse source code across different platforms, X is unusual in its portability across software build environments. This paper describes several mechanisms used in the MIT release of the X Window System to obtain such flexibility, and summarizes some of the lessons learned in trying to support X on a number of different platforms.

## 1. Introduction

The X Window System is a portable, network transparent window system originally developed at MIT. It is intended for use on raster display devices ranging from simple monochrome frame buffers to deep, true color graphics processors. Because of its client/server architecture, the non-proprietary nature of its background, and the portability of the sample implementation from MIT, the X Window System has rapidly grown to become an industry standard. This portability is the result of several factors: a system architecture that isolates operating system and device-specifics at several levels; a slow, but machine-independent, graphics package that may be used for an initial port and to handle cases that the underlying graphics hardware does not support; and the use of a few, higher-level tools for managing the build process itself.

### 1.1. Summary of X Window System Architecture

The X Window System is the result of a combined effort between MIT Project Athena and the MIT Laboratory for Computer Science. Since its inception in 1984, X has been redesigned three times, culminating in Version 11 which has since become an industry standard (see [Scheifler 88] for a more detailed history). X uses the client/server model of limiting interactions with the physical display hardware to a single program (the *server*) and providing a way for applications (the *clients*) to send messages (known as *requests*) to the server to ask it to perform graphics operations on the client's behalf. These messages are sent along a reliable, sequenced, duplex byte stream using whatever underlying transport mechanisms the operating system provides. If connections using network virtual circuits

† X Window System is a trademark of MIT; DECnet is a trademark of Digital Equipment Corporation; UNIX is a registered trademark of AT&T.

(such as TCP/IP or DECnet) are supported, clients may be run on any remote machine (including ones of differing architectures) while still displaying on the local server.

The details of how the client establishes and maintains connections with the server are typically hidden in a subroutine package (known as a *language binding*) which provides a function call interface to the X protocol. Higher level toolkits and user interface management systems are then built on top of the binding library, as shown in Figure 1 for the C programming language. Since only the underlying operating system networking interface of the binding (shown in *italics*) need be changed when porting to a new platform, well-written applications can simply be recompiled.

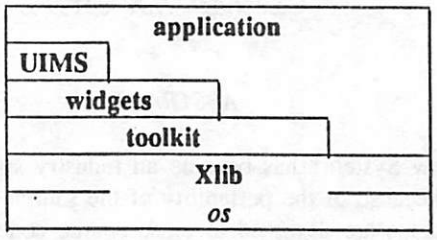


**Figure 1:** architecture of a typical C language client program

The server takes care of clipping of graphics output and routing keyboard and pointer input to the appropriate applications. Unlike many previous window systems, moving and resizing of windows are handled outside the server by special X applications called *window managers.* Different user interface policies can be selected simply by running a different window manager.

The MIT sample server can be divided into three sections: a device-independent layer called *diX* for managing the various shared resources (windows, pixmaps, colormaps, fonts, cursors, etc.), an operating system layer called *os* for performing machine-specific operations (managing connections to clients, dealing with timers, reading color and font name databases, and memory allocation), and a device-specific layer called *ddX* for drawing on the display and getting input from the keyboard and pointer. Only the *os* and *ddX* portions of the server need to be changed when porting X to a new device.

Although this is still a substantial amount of work, a collection of pixel-oriented drawing packages that only require device-specific routines (refered to as *spans*) to read and write rows of pixels are provided to allow initial ports of X to be done in a very short time. A server developer can then concentrate on replacing those operations that can be implemented more efficiently by the hardware. Figure 2 shows the relative layering of the various packages within the sample server from MIT. The *mi* library provides highly portable, machine-independent routines that may be used on a wide variety of displays. The *mfb* and *cfb* libraries contain versions of the graphics routines for monochrome and color frame buffers, respectively. Finally, the *snf* library can be used to read fonts stored in Server Natural Format. Typically, only the sections printed in *italics* need be changed when moving to a new platform.

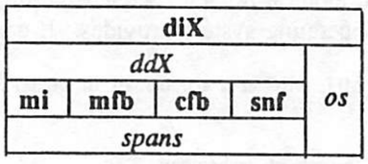


**Figure 2:** architecture of the MIT sample server

By splitting out the device-specific code (by separating clients from servers and *diX* from *ddX*) and then providing portable utility libraries (*mi*, *mfb*, *cfb*, and *cfb*) that may be used to implement the nonportable portions of the system, much of the code can be reused across many platforms, ranging from personal computers to supercomputers.

## 2. Configuring the Software Build Process

In practice, porting X to a new platform typically requires adding support in the operating system-specific networking routines and mixing together pieces of machine-independent and device-specific code to access the input and output hardware. Although this approach is very portable, it increases the complexity of the build process as different implementations require different subsets. One solution is to litter the source code with machine-specific compiler directives controlling which modules areas get built on a given platform. However, this rapidly leads to sources that are hard to understand and even harder to maintain.

A more serious problem with this approach is that it requires configuration information to be replicated in almost every module. In addition to being highly prone to error, modifying or adding a new configuration becomes extremely difficult. In contrast, collecting the various options and parameters in a single location makes it possible for someone to reconfigure the system without having to understand how all of the modules fit together.

Although sophisticated software management systems are very useful, they tend to be found only on specific platforms. Since the configuration system must be working before a build can begin, the MIT releases try to adhere to the following principles:

- Use existing tools to do the build (e.g. *make*) where possible; writing complicated new tools simply adds to the amount of software that has to be bootstrapped.
- Keep it simple. Every platform has a different set of extensions and bugs. Plan for the least common denominator by only using the core features of known tools; don't rely on vendor-specific features.
- Providing sample implementations of simple tools that are not available on all platforms (e.g. a BSD-compatible *install* script for System V) is very useful.
- Machine-dependencies should be centralized to make reconfiguration easy.
- Site-wide options (e.g. default parameters such as directory names, file permissions, and enabling particular features) should be stored in only one location.
- Rebuilding within the source tree without losing any of the configuration information must be simple.
- It should be possible to configure external software without requiring access to the source tree.

One approach is to add certain programming constructs (particularly conditionals and iterators) to the utility used to actually build the software (usually *make*; see [Lord 88]). Although this an attractive solution, limits on time and personnel made implementing and maintaining such a system impractical for X.

The MIT releases of X employ a less ambitious approach that uses existing tools (particularly *make* and *cpp*). *Makefiles* are generated automatically by a small, very simple program named *imake* (written by Todd Brunhoff of Tektronix) that combines a template listing variables and rules that are common to all *Makefiles*, a machine- and a site-specific configuration file, a set of rule functions written as *cpp* macros, and simple specifications of targets and sources called *Imakefiles*. Since the descriptions of the inputs and outputs of the build are separated from the commands that implement them, machine dependencies such as the following can be controlled from a single location:

- Some versions of *make* require that the variable SHELL to be set to the name of the shell that should be used to execute *make* commands.
- The names of various special *make* variables (e.g. MFLAGS vs. MAKEFLAGS) differ between versions.
- Special directives to control interaction with source code maintenance systems are required by some versions of *make*.

- Rules for building targets (e.g. *ranlib*, lint options, executable shell scripts, selecting alternate compilers) differ among platforms.
- Some systems require special compiler options (e.g. increased internal table sizes, floating point options) for even simple programs.
- Some systems require extra libraries when linking programs.
- Not all systems need to compile all sources.
- Configuration parameters may need to be passed to some (such as -DDNETCONN to compile in DECnet support) or all (such as -DSYSV to select System V code) programs as preprocessor symbols.
- Almost all systems organize header files differently, making static dependencies in *Makefiles* impossible to generate.

By using the C preprocessor, *imake* provides a familiar set of interfaces to conditionals, macros, and symbolic constants. Common operations, such as compiling programs, creating libraries, creating shell scripts, and managing subdirectories, can be described in a concise, simple way. Figure 3 shows the *Imakefile* used to build a manual page browser named *xman* (written by Chris Peterson program of the MIT X Consortium, based on an implementation for X10 by Barry Shein):

```
DEFINES = -DHELPFILE=\"$(LIBDIR)$(PATHSEP)xman.help\"
LOCAL_LIBRARIES = $(XAWLIB) $(XMULIB) $(XTOOLLIB) $(XLIB)
SRCS = ScrollByL.c handler.c man.c pages.c buttons.c help.c menu.c search.c \
       globals.c main.c misc.c tkfuncs.c
OBJS = ScrollByL.o handler.o man.o pages.o buttons.o help.o menu.o search.o \
       globals.o main.o misc.o tkfuncs.o
INCLUDES = -I$(TOOLKITSRC) -I$(TOP)

ComplexProgramTarget (xman)
InstallNonExec (xman.help, $(LIBDIR))
```

**Figure 3:** *Imakefile* **used by a typical client program**

This application requires the name of the directory in which its help file is installed (which is a configuration parameter), several libraries, and various X header files. The macro *ComplexProgramTarget* generates the appropriate rules to build the program, install it, compute dependencies, and remove old versions of the program and its object files. The *InstallNonExec* macro generates rules to install *xman*'s help file with appropriate permissions.

## 3. Generating Makefiles

Although *imake* is a fairly powerful tool, it is a very simple program. All of the real work is performed by the template, rule, and configuration files. The version currently used at MIT (which differs somewhat from the version supplied in the last release of X) uses symbolic constants for all configuration parameters so that they may be overridden or used by other parameters. General build issues (such as the command to execute to run the compiler) are isolated from X issues (such as where should application default files be installed) by splitting the template as shown in Figure 4.

This template instructs *imake* to perform the following steps when creating a *Makefile*:

1. Using conditionals, *Imake.tmpl* determines the machine for which the build is being configured and includes a machine-specific configuration file (usually named *machine*.cf). Using the C preprocessor to define various symbols, this configuration file sets the major and minor version numbers of the operating system, the names of any servers to build, and any special programs (such as alternate compilers) or options (usually to increase internal table sizes) that need to be used during the build. Defaults are provided for all parameters,

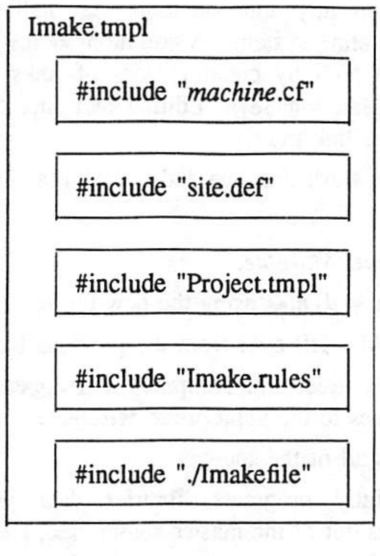


**Figure 4:** structure of *imake* template used by X

so .cf files need only describe how this particular platform differs from ''generic'' UNIX System V or BSD UNIX. Unlike previous versions of the *imake* configuration files, when new parameters are added, only the systems which are effected by them need to be updated.

2. Next, a site-specific file (named *site.def*) is included so that parameters from the .cf files may be overridden or defaults for other options provided. This is typically used by a site administrator to set the names of the various directories into which the software should be installed. Again, all of the standard *cpp* constructs may be used.
3. A project-specific file (named *Project.tmpl*) is included to set various parameters used by the particular software package being configured. By separating the project parameters (such as directories, options, etc.) from build parameters (such as compilers, utilities, etc.), the master template and the .cf files can be shared among various development efforts.
4. A file containing the set of *cpp* rules (named *Imake.rules*) is included. This is where the various macro functions used in the master template and the per-directory description files (named *Imakefile*) are defined. These rules typically make very heavy use of the *make* variables defined in *Imake.tmpl* so that a build's configuration may be changed without having to edit this file.
5. The *Imakefile* describing the input files and output targets for the current directory is included. This file is supplied by the programmer instead of a *Makefile*. The functions that it invokes are translated by *cpp* into series of *make* rules and targets.
6. Finally, *make* rules for recreating the *Makefile* and managing subdirectories are appended, and the result is written out as the new *Makefile*.

*Imake*, along with a separate tool (named *makedepend*, also written by Brunhoff) that generates *Makefile* dependencies between object files and the source files used to build them, allows properly configured *Makefiles* to be regenerated quickly and correctly. By isolating the machine- and site-specifics from the programmer, *imake* is much like a well-developed text formatter: both allow the writer to concentrate on the content, rather than the production, of a document.

## 4. How X uses *imake*

Development of X at MIT is currently done on more than half a dozen different platforms, each of which is running a different operating system. A common source pool is shared across those machines that support symbolic links and NFS by creating trees of links pointing back to the master sources (similar to the object trees of [Harrison 88]). Editing and source code control is done in the master sources and builds are done in the link trees.

A full build is done by creating a fresh link tree and invoking a simple, stub top-level *Makefile* which:

1. compiles *imake.*
2. builds the real top-level *Makefile.*
3. builds the rest of the *Makefiles* using the new top-level *Makefile.*
4. removes any object files left over from the previous build.
5. builds the header file tree, and computes and appends the list of dependencies between object files and sources to the appropriate *Makefiles.*
6. and finally, compiles all of the sources.

If the build completes successfully, programs, libraries, data files, and manual pages may then be installed. By keeping object files out of the master source tree, backups and releases can be done easily and efficiently. By substituting local copies of particular files for the appropriate links, developers can work without disturbing others.

## 5. Limitations

Although the system described here is very useful, it isn't perfect. Differences between utilities on various systems places a restriction on how well existing tools can be used. One of the reasons why *imake* is a program instead of a trivial invocation of the C preprocessor is that some *cpp*'s collapse tabs into spaces while others do not. Since *make* uses tabs to separate commands from targets, *imake* must sometimes reformat the output from *cpp* so that a valid *Makefile* is generated.

Since *cpp* only provides global scoping of symbolic constants, parameters are visible to the whole configuration system. For larger projects, this approach will probably prove unwieldy both to the people trying to maintain them and to the preprocessors that keep the entire symbol table in memory.

The macro facility provided by *cpp* is convenient because it is available on every platform and it is familar to most people. However, a better language with real programming constructs might provide a better interface. The notions of describing one platform in terms of another and providing private configuration parameters map intriguingly well into the models used in object management systems.

## 6. Summary and Observations

The sample implementation of the X Window System from MIT takes advantage of a system architecture that goes to great lengths to isolate device-dependencies. By selectively using portable versions of the device-specific functions, a developer moving X to a new platform can quickly get an initial port up and running very quickly.

To manage the various combinations of modules and to cope with the differing requirements of every platform and site, X uses a utility named *imake* to separate the description of sources and targets from the details of how the software is actually built. Using as few external tools as possible, this mechanism allows support for new platforms to be added with relatively little effort.

Although the approaches taken by MIT will not work for everyone, several of its experiences may be useful in other projects:

- Even if portability isn't a goal now, it probably will become one sooner than expected.
- Just as in other areas, it frequently pays to periodically stand back from a problem and see whether or not a simple tool will help. With luck and the right amount of abstracting it may even solve several problems at once.

- Be wary of anything that requires manual intervention.
- And finally, there is no such thing as portable software, only software that has been ported.

## 7. References

[Harrison 88]
"Rtools: Tools for Software Management in a Distributed Computing Environment," Helen E. Harrison, Stephen P. Schaefer, Terry S. Yoo, *Proceedings of the Usenix Association Summer Conference*, June 1988, 85-94.

[Lord 88]
"Tools and Policies for the Hierarchical Management of Source Code Development," Thomas Lord, *Proceedings of the Usenix Association Summer Conference*, June 1988, 95-106.

[Scheifler 88]
*X Window System: C Library and Protocol Reference*, Robert Scheifler, James Gettys, and Ron Newman, Digital Press, Bedford, MA, 1988.

# Automating the importation of software

*Paul R. Eggert*
West Coast Research Center
Unisys Corporation
2400 Colorado Av, Santa Monica, CA 90406, USA

Author's current address:
Twin Sun, Inc.
360 N. Sepulveda, El Segundo, CA 90245, USA

**Abstract.** An obstacle to reusing outside software is the overhead of maintaining its tailoring and installation, especially in heterogeneous networks with many hosts. Conventional installation procedures are designed by software *exporters;* we describe a complementary approach for use by *importers.* In this approach, software objects are periodically rebuilt and installed from imported sources automatically. The imported sources are not changed, and identical sources are maintained on all hosts. Automated rebuilding, combined with source control, enforces disciplined maintenance of tailoring and installing reused software. It helps replace irreproducible installations by codified, tested rebuilding procedures.

## 1. Problem: maintaining reused tools

A major obstacle to reusing software tools developed by others is the overhead of maintaining their tailoring and installation. One would like to hand a tool to one's software engineering environment and say, "Install this in the right places." In practice, however, a human installer must often tailor some sources for local conventions. And when the supplied installation procedure fails, as it often does, the installer must patch the procedure or even finish the installation by hand. The result is an irreproducible installation. If a new version of the tool arrives, or if the tool needs installation on another host, or if a change to the underlying environment mandates rebuilding the tool, the manual work must be redone.

In a larger organization, installation is often maintained by several support personnel, who come to be associated with the tools they install. When a support person migrates, the organization loses installation expertise. Worse, with heterogeneous hosts, each manual installation must be redone for each host type, each time perhaps slightly differently, and the problem is multiplied. Commercial software organizations often expend great efforts debugging installation procedures to help ease software export. We describe the opposite approach: maintaining and testing software *import* procedures.

## 2. The rebuilder's goals

For the past two years we have used a "rebuilder", an automated procedure for maintaining software tools that supports the following goals:

- Rebuild all software tools from sources fully automatically.
- Reinstall all tools on running systems fully automatically.
- Reuse foreign software without changing its sources.
- Maintain identical sources on all hosts.
- Periodically test all rebuilding procedures.

The terms "sources" and "objects" here mean something other than the usual notions of source code and object code (or object oriented programming). *Objects* are built automatically from sources by the rebuilder. *Sources* are everything else, namely, everything maintained locally by hand. These terms will always be used from the local viewpoint. For example, executable binaries supplied on magnetic tape by a foreign organization are sources, because they cannot be rebuilt automatically but must instead be maintained by hand as new releases arrive. Conversely, a C language file is an object if it is the output of a Lisp-to-C translator. Every software item is either source or an object, but not both. Only objects are *installed*, i.e. made available to users in a conventional location.

## 3. The rebuilder's actions

The rebuilder is a collection of software tools that currently runs under Unix. For brevity, the following discussion assumes understanding of Unix software engineering practice.

Each tool's sources are kept in a separate directory. Many tools are so complicated that their source directories have subdirectories; this complication does not affect the rebuilder and will be ignored hereafter. Source directories form a hierarchy organized by host type. For example, if the hierarchy is rooted at src, src/etc is distinguished from src/usr, because every host must rebuild /etc, but diskless hosts should not rebuild src/usr because they read their fileservers' /usr.

The rebuilder operates in several phases. If a phase fails, later phases are not attempted; instead, the rebuilder reports failure to its human supervisors via electronic mail. Each phase operates by traversing the source hierarchy in preorder. Internal nodes of the source tree tell the rebuilder what subtrees are needed on the rebuilding host type, using a subsidiary Unix shell file Need. In the above example, the command src/Need etc always succeeds, whereas src/Need usr succeeds only on a host with its own copy of /usr. The simplest leaves of the source tree are single source files or manual pages, which the rebuilder simply compiles and installs into the local binary or manual page directories. More commonly, a leaf is a directory containing a makefile, which the rebuilder handles with the Unix command make [2].

The rebuilder's phases are characterized by the arguments passed to make at each leaf:

make clean
: Remove objects in the source hierarchy left over from previous builds.

make diff
: Build objects from sources, and report differences between installed objects and objects newly rebuilt.

make install
: Install any changed objects.

The last phase is trickiest, because it may affect or replace currently executing programs. Ideally, the make install phase should be atomic, and old sessions should continue to run as if no change had occurred. In practice, most changes are applied to a new, temporary directory; at the end it is renamed into the proper place as atomically as possible. Installations that affect operation of the underlying environment are treated even more carefully. For example, make install in src/etc can change the list of serial devices, so the makefile in src/etc propagates any changes to Unix's init process by issuing the required kill −1 1 command.

## 4. Reusing foreign sources

To incorporate a new tool, we create a new leaf directory in the source hierarchy and place the new sources under this leaf. If the sources are foreign, we maintain local tailoring in separate files. Usually it is enough to create a local makefile with a special name; the rebuilder prefers such a makefile, passing it to a make preprocessor (beyond the scope of this paper). Sometimes more than just makefile changes are needed; in effect, local edits to foreign sources must be maintained. For example, GNU Emacs [3], with about 8000 Kbytes of sources, requires about 9 Kbytes (400 lines) of tailoring sources. The local makefile helps distinguish local tailoring from foreign sources.

All sources are a mixture of local and foreign. Even locally developed sources can be thought of as sources that tailor nothing. We avoid rework and encourage further reuse by minimizing and compartmentalizing local tailoring.

## 5. Identical sources on all hosts

Because the rebuilder is completely automatic, each host has identical sources. Otherwise manual intervention would be needed for host dependent tailoring. Identical sources require automating manual flags. For example, instead of maintaining a source file d.h containing the lines

```
/* Set iff your OS has termio. */
#define hasTermio 0
```

we can maintain a portable shell file that compiles a simple test program testing for termio's presence, and outputs a corresponding line into d.h; d.h becomes an object instead of source. A similar approach is found in Wall's metaconfigurer [5]. Wall's system encourages reuse of portable configuration files like the shell file described above, but unfortunately it also requires manual intervention for many configurations.

## 6. Periodic rebuilding

One must test local sources by periodically running the rebuilder, thus reinstalling all locally maintained objects from the sources. Do this too often, and too many resources will be consumed; do it too rarely, and installation will go untested for too long. The rebuilder can be run selectively on a subtree of the sources if a change's effect must be propagated immediately. Periodic automated rebuilding helps insure that this is the only effective way to alter installed objects. We rebuild everything once daily, because we are constantly developing software.

## 7. Change control

Source change control à la RCS [4] or SCCS [1] is essential for recording local changes and communicating them to other developers and installers. We extended RCS slightly to mail source change logs daily to interested parties. With the morning coffee, one can study yesterday's source changes, followed by last night's report of changes to installed objects. Letting $S$ denote the sources, $O$ the installed objects, and $S(O)$ the installed objects resulting from rebuilding the sources, then the rebuilder should compute a fixed point $O$ of $S$ such that $O=S(O)$. Successive reports that $S$ stayed the same but $O$ changed suggest manual intervention.

## 8. Problems

Rebuilding everything from scratch takes both CPU and disk resources. For example, we build and install about 25 Mbytes of objects from about 35 Mbytes of sources in under five hours on a single Sun-3/160; other Suns share or copy the results. Had we more sources, we would run the rebuilder in parallel: our workstation network has much spare CPU capacity at night.

Most rebuild failures occur because of errors during the make clean and make diff passes before any changes propagate to installed code. We avoid disk space exhaustion during installation by careful (though imperfect) preallocation. There is always the possibility of an installation disaster that a human would recognize and abort, but the rebuilder would ignore and continue. Should this happen, we would try to prevent recurrence of the particular disaster by adding extra checks. It has not yet happened to us, and the rebuilder's advantages outweigh this small risk.

One problem requiring more thought is distributing the rebuilder, as well as software built under the rebuilder. Because the rebuilder is built using itself, bootstrapping issues arise. We would like to test bootstrapping on each rebuild, not just on rebuilds done elsewhere. This problem interacts with change control on multiple copies of the sources.

## 9. Conclusion

Rebuilding things fully automatically is a significant advance over ordinary software engineering practice, just as using make was an advance over the old practice of doing all compilations by hand. It is hard to imagine reverting to the old, undisciplined days of manual installation. One measure of the confidence we place in the rebuilder is that when a new release of the operating system arrives, we often simply discard the old system partitions, bring in the new vanilla system, and run the rebuilder. The only files that need saving are user mailboxes and the password file. If we could remove these exceptions, we would never need to dump the system partition!

As software reuse grows, so will the cost of reinstallation. Automated rebuilding, combined with source code control, enforces disciplined maintenance of tailoring and installing reused software. It helps replace irreproducible installations by codified, tested rebuilding procedures. It is more work the first time, but it saves work later and helps improve the quality of reused software.

## 10. References

1. Eric Allman, *An Introduction to the Source Code Control System,* UC Berkeley 4.3BSD, 1980 December 5.
2. S. I. Feldman, *Make—A Program for Maintaining Computer Programs,* UC Berkeley 4.3BSD, 1986 April.
3. Richard M. Stallman, *GNU Emacs 18.53,* Free Software Foundation, Cambridge, MA, 1989 February 24.
4. Walter F. Tichy, Design, Implementation, and Evaluation of a Revision Control System, *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, 1982 September.
5. Larry Wall, *Dist Kit 1.0,* System Development Corporation, A Burroughs Corporation, Santa Monica, CA, 1987 May 22.

# The Use of a Time Machine to Control Software

*Andrew G. Hume*

research!andrew
andrew@research.att.com

*ABSTRACT*

The Unix system has always supported a good environment for programming. The main ingredients are a uniform (if low-level) view of the filesystem with files as sequences of bytes, a rich set of diverse user-level tools that apply to byte streams, and mechanisms such as pipes for connecting processes together.

This paper describes how an exotic (special purpose) filesystem designed for browsing through the files in a backup system can be useful in software development. Various other exotic filesystems applicable to software development are also described.

## Introduction

This paper discusses the use of exotic filesystems in software development. Exotic filesystems are a mature widespread technology largely ignored by the software engineering community. I will discuss the filesystems in the context of a couple of simple, common software development problems. Of course, exotic filesystems are useful in other contexts; maybe you will think of some yourself.

The setting is a common one; you have a largish file tree of source and makefiles. Using a small set of system utilities such as a C compiler, system header files and libraries, these sources are compiled and linked into executables which are then installed (typically) outside the source tree.

Surprisingly, people still do business this way (including the Research version of the Unix system). The most common improvement to this model is adding source version control. The oldest and probably the most common source control system is SCCS[1]. There are other systems such as RCS[2] but for our purposes they are more or less equivalent to SCCS. Version control complicates our model in two ways; in principle we have another tree of version history files from which we generate our source tree, and we have to have either an explicit or implicit (use the latest version) list which maps source file and version number.

Another independent improvement is to allow developers to duplicate and work on a small piece of the source tree and somehow map the missing parts of the source tree to the corresponding part of the main or reference source tree. This has been commonly done by a technique called *viewpathing* which says that if we refer to a file *foo.c* , the system should search through a specified list of directories for that file. The canonical example is a list of a developer's work area, the group's work area, and lastly the company's main reference release area.

Of course, there are many other improvements described in the software engineering literature. Most tend towards more tightly integrated worlds, implemented on but largely divorced from the native Unix system.

**Some Problems**

*How can multiple developers work on the same source simultaneously?*

In general, multiple developers working in the same directory(s) will interfere with each other, either by competing for a source file or by using object files currently under development. The next most obvious step is to give each developer a copy of the source tree and work in that copy. The main drawbacks are setting up the copies (it takes time to copy and bring to equilibrium by doing re-compiling, etc.), disk usage (few interesting systems are small enough to copy more than a few times), keeping the copies up to date (a tractable, tedious problem although the relevant tools are not common), and merging the updates back into the reference source tree (the problem combining two updates to the same file is difficult enough that most systems avoid it by fiat).

The next step drew from the idea of a search path for executing programs. One form of the *exec* system call specifies a list of directories which are searched in order for an executable file of the given name. The first one found is executed. There is an obvious analogy for the source problem. If software knew to search for files in the local work area and then in the reference system, the local copy would only have to contain locally modified source and the corresponding object files. There is no operating system support for this so this scheme has to be implemented at user level (a white lie; exceptions will be described below). There are several ways to do this; the simplest involves changing just one tool — *make* [3]. The best known example is *build* [4]. The idea is simple: define an analog to the execute search path called the *viewpath*. When *build* looks for a file to use in a command recipe, it searches through this list and substitutes the full pathname for the file's name in the recipe. This works nearly all the time; some programmer cooperation, as described by Lord[5], is needed to make header files work properly. Another descendant of *make* to implement viewpathing is *nmake* [6].

It should be noted that *build* cannot just compare the modification times of source and object files to determine if they are out of date. The problem arises if the source name maps to different files in different calls to *build*. For example, a programmer may have a local copy of list.c with debugging. After discovering the bug is elsewhere, the local copy list.c is removed. The next invocation of *build* sees a fresh copy of list.o more recent than the (reference) source and does not rebuild it. The solution is simple; use a database to record the actual dependencies using full pathnames and compare the mapped names to the names in the database. Another disadvantage of this scheme is that it is harder to understand exactly how a piece of software has been made; it depends very much on the viewpath (which is typically carried around in the environment) and on people adding or deleting files from any directory in that viewpath.

*How do you snapshot or store a release?*

This is a difficult problem because there are several solutions with varying degrees of convenience. Let's start with a real and very paranoid example. AT&T is in the business of selling and maintaining phone switches (amongst other things). Some of these switches are very old, running software generated on machines that no longer exist (got a PDP9, anyone?). How do you fix bugs in the switching software? (We will ignore the financially correct answer, which is to give the customer a free upgrade to something that can be fixed.) You take a complete disk dump of the system when the software was generated and thus can regenerate the software anytime you have the hardware. And yes, Virginia, when the hardware goes away, you write a simulator. This method may require lots of space to store the disk dump.

Another, much more common, method is to simply capture the source tree. This may involve a physical copy (say on tape) or simply a list of version numbers. In any case, we assume that the objects can be regenerated from the source using the current system. This wrong in at least two important ways. Firstly, you have missed a whole bunch of source in the form of system header files (say those in /usr/include) and binaries such as system libraries. You are also assuming a lot about the constancy of system tools used in making or processing the source such as *make*, *awk*, *yacc*, and *lex*. There may also be missing source in that someone may use files and/or data outside the source tree. Secondly, it is an act of faith that two different compilers will produce the same program from identical source. If your needs are more stringent — the program must fit in a 4K PROM — you need to be much more optimistic.

I think most people would like the reassurance of the former method but find the inconveniences involved in making, storing and using the dumps too much and revert to the latter method. This suggests an intermediate solution; capture the source and binaries (compilers, libraries, etc.) that you believe are necessary and sufficient to build your software on an otherwise empty filesystem and then rebuild your software after isolating yourself to that filesystem via the *chroot* system call.

With any of these techniques it is still hard to investigate or examine source between different releases. For example, you may want to examine how a module changed over time, how a directory of source changed over time or how groups of files change as a unit.

**Remote Filesystems**

Before I describe how some special-purpose filesystems can address these problems, I will briefly discuss remote filesystems.

The central concept underpinning remote filesystems is the filesystem switch. This is exactly analogous to the switch for character and block devices in the Unix kernel. Access to the filesystem is restricted to a small number (10) of functions and these are referenced through a table indexed by filesystem type. This was first done by Weinberger[7] for Eighth Edition Unix around 1982. (Another implementation using vnodes and 24 functions was done later at Sun by Kleiman[8].) Once you have a filesystem switch, there are roughly two ways to implement a new type of filesystem. The first is to write filesystem code inside the kernel; this is how the normal filesystem is done and how /proc was implemented. /proc is a filesystem written by Killian[9] that makes running processes visible in the filesystem. *Ps* can now work by reading directories and files rather than grubbing around in the kernel's memory and debuggers such as *pi* can read and alter a process's state and memory by doing read, writes and ioctl's on a file.

The second way to add new filesystems is the motivation for Weinberger's work: you write stub routines that map the filesystem switch calls into request/response packets sent over a stream (or socket). The stream may be, and often is, connected to a server process on another machine. Sun's NFS filesystems are done in a similar way; the filesystem semantics are slightly different and the protocol describing the packets sent over the stream is much more complicated. System V has another similar system called RFS[10]. The server is free to assign any meaning it likes to the requests made to it. A debugging filesystem, for example, could treat procedures names as directories and variable names as entries in those directories.

There are of course other ways to do remote filesystems. The Newcastle Connection is a remote filesystem done at user level by recompiling programs with a special library. This has the advantage of being easy to implement, user level code is easier to do and debug than kernel code. It has the disadvantages of having to recompile your programs and it does miss some of the normal filesystem semantics (such as where does a program drop core?).

**Exotic Filesystems**

As of the time of writing (1989), remote filesystems have been around for about seven years. The Ninth Edition Unix has been running *netb* (the second version of Weinberger's filesystem) for about two years. Sun's NFS has been in wide use for more than 3 years; AT&T's System V RFS has been out for nearly as long. During this time, what types of filesystems have been implemented? Obviously, the most common answer is a filesystem reflecting regular filesystems on a remote machine. What is astonishing to me is that it is just about the only answer. The published answers known to me are*

- /proc — the Eighth Edition process filesystem by Killian described above.
- MFS — a mail filesystem by Hitz and Honeyman[12] on Eight Edition Unix at Princeton. The idea is that cat > /mail/research/andrew sends mail to research!andrew.
- a face server by Pike and Presotto[13] for visual signaling of incoming mail on Eighth Edition Unix.
- a backup filesystem — a part of the File Motel by Hume[14] on Eighth/Ninth Edition Unix. The idea here is the filesystem is based on a database and optical jukebox and presents a filesystem as it was at arbitrary times in the past.
- TFS — a translucent filesystem at Sun via NFS that allows multiple directories to be mounted on a name.
- *gcan* — an NFS implementation of the Berkeley Unix system *restore* program by Bill Roome at AT&T Bell Laboratories. Given a number of full and incremental dumps, it presents a filesystem that can move through time (like the above backup filesystem).

**Some Filesystem Solutions**

*How can multiple developers work on the same source simultaneously?*

The most obvious way for exotic filesystems to help is to implement some form viewpathing at the filesystem level so that all tools get to use it. The technique most used is to allow multiple directories to be mounted at a single name with the idea that entries in earlier directories are overridden by entries with the same name in later directories. (You can then remove the execute search path stuff from the *exec* system call.) Both the Plan9 filesystem and TFS support this notion. The Plan9 filesystem is vulnerable to the same problem of removing files from intermediate directories that viewpathing has. TFS explicitly prevents this from happening; when you remove a file and this would cause a another entry to become visible, TFS records the entry name with a "whiteout" so it will never appear again (unless you create an entry with that name). For our purposes, you would have to use a special command that would allow TFS to show the underlying entry.

The main advantage of exotic filesystems here is that all the normal system tools, such as debuggers, see the effects of viewpathing rather than just isolated tools such as *make*. The notion of viewpathing can also be combined with version control as in the 3—D filesystem described below.

*How do you snapshot or store a release?*

There are two obvious ways that exotic filesystems can help, depending on whether you use version control. If you don't, then the best way to help is to provide a historical filesystem. The two known to me, *gcan* and the backup filesystem, are somewhat similar in use; the major difference is in the underlying implementation. *Gcan* uses regular dumps for its data and supports the NFS file server protocol. Both systems solve the problem the same way: allow the user to make the filesystem appear as it did as of the instant of release.

The File Motel is an incremental backup system using a database to provide quick access to any file that has been backed up. Typically, the files are saved to archival media such as Write—Once—Read—Many optical disks stored in a autochanger or jukebox, eliminating any human operator intervention. The motivation for the File Motel was immediate access to prior versions of a file. To facilitate browsing the versions stored, the File Motel supports a time-varying file system. While this does not seem overtly useful to software management, it supports a fairly complete form of version control. The idea is

*there is considerable activity in filesystems in industry, particularly at AT&T and Sun, that is not published or not widely known.

that rather than say we need to look at the software at state T which is a very large tuple of version numbers, we can say what did our filesystem look like at time *t*? We now no longer care about whether some list of the tools, headers or any other system resource is complete; we have the whole file system. Of course, version numbers are still useful, particularly for identifying the pedigree of objects out in the field.

If you use version control, then use a filesystem that supports this directly. This can come in two flavours. The most direct is where versions are denoted explicitly, such as in src/access.c/2.1. You would probably want an easy way (such as a missing version number) to refer to the most recent version. In this scheme, the explicit versions used in a product are used by the relevant makefiles directly. In the other flavour, the version numbers are not available explicitly but are stored in a file used by the file server. The 3-D filesystem by Korn and Krell[15] provides this kind of filesystem, although it does it at user level through special libraries.

In both cases, the filesystem not only helps by providing uniform access to the versioned source for all tools, but it can ease some of the burden of using a version control system. For example, when a developer writes a file, the filesystem can create a new version automatically or warn that another developer is working on that file.

## Summary

The technology for implementing exotic filesystems is commonly available and not very hard to use. Exotic filesystems can directly help some of the steps involved in software development. By working through the filesystem level, the advantages leverage to nearly all tools used in software development. A few such filesystems have been implemented and have been described above. There are many others awaiting discovery; let's get out there and get'em!

## Acknowledgements

While the content of this paper is my responsibility, David Tilbrook goaded me into writing it. Some of the material on remote filesystems came from people responding to a request on netnews. There were too many people to name here, but special mention must go Bruce Ellis and Bill Shannon. Peter Weinberger deserves praise for devising and implementing a useful file server protocol simple enough for others to use but he doesn't get it from me because he never published a description of it.

## References

[1] Rochkind, M. J., "The source code control system", *IEEE Trans. Software Engineering* , **SE-1**(4), pp364—370, 1975.

[2] Tichy, W. F., "RCS — A System for Version Control", *Software Practice & Experience* , **15**(7), pp637—654, 1985.

[3] Feldman, S. I., "Make — a program for maintaining computer programs", *Software Practice & Experience* , **9**, pp255—265, 1979.

[4 Erickson, V. B., and Pellegrin, J. F., "Build, A Software Construction Tool", *AT&T Bell Laboratories Technical Journal* , **63**(6), pp1049—1059, July 1984.

[5] Lord, T., "Tools and Policies for the Hierarchical Management of Source Code Development", *USENIX San Francisco 1988 Summer Conference Proceedings* , pp95—106, 1988.

[6] Fowler, G. S., "The Fourth Generation Make", *USENIX Portland 1985 Summer Conference Proceedings* , pp159—174, 1985.

[7] Weinberger, P., "The version 8 network file system (abstract)", *USENIX Salt Lake City 1984 Summer Conference Proceedings* , p86, 1984.

[8] Kleiman, S. R., "Vnodes: An Architecture for Multiple File System Types in SUn UNIX", *USENIX Atlanta 1986 Summer Conference Proceedings* , pp238—247, 1986.

[9] Killian, T. J., "Processes as Files", *USENIX Salt Lake City 1984 Summer Conference Proceedings* , p203—207, 1984.

[10] Rifken, A. P., Forbes, M. P., Hamilton, R. L., Sabrio, M., Shah, S., and Yeuh, K., "RFS Architectural Overview", *USENIX Atlanta 1986 Summer Conference Proceedings* , pp248—259, 1986.

[12] Hitz, D., and Honeyman, P., "A Mail File System for Eight Edition UNIX", *USENIX Atlanta 1986 Summer Conference Proceedings* , pp391—394, 1986.

[13] Pike, R., and Presotto, D. L., "Face The Nation", *USENIX Portland 1985 Summer Conference Proceedings* , pp81—85, 1985.

[14] Hume, A. G., "The File Motel — An Incremental Backup System for UNIX", *USENIX San Francisco 1988 Summer Conference Proceedings* , pp95—106, 1988.

[15] Korn, D. G., and Krell, E., "Transparent Version Control for the Unix System", *Proceedings of the VIII International Conference of the Chilean Computer Society* , July 1988, IEEE.

# Experiences Using a Hypertext Framework to Manage Software

*Peter Nicklin*
nicklin%hpdtc@hplabs.hp.com

Hewlett-Packard
5301 Stevens Creek Boulevard
Santa Clara, California 95052

*ABSTRACT*

The SPMS Software Project Management System is a tool for creating and traversing networks of typed symbolic links superimposed on the UNIX™ directory hierarchy. These networks provide a powerful way of managing the development of complex software applications. This paper describes the techniques developed and experience gained at Hewlett-Packard in using SPMS to manage the development of a custom VLSI design system called Chipbuster.

## Introduction

As computers increase in power, we use them to perform more complex tasks that require more complex software applications to be written and managed. This paper describes our experience in using a hypertext framework to manage a complex UNIX-based application for custom VLSI design called Chipbuster.

We use the SPMS Software Project Management System[1] to manage the Chipbuster software. SPMS was released as contributed software with the 4.2BSD Berkeley Software Distribution in 1983. It is a tool for creating and traversing networks of typed symbolic links superimposed on the UNIX directory hierarchy.

SPMS shares the concept of typed links with hypertext systems such as Brown University's Intermedia[2], Xerox PARC's NoteCards[3], and Tektronix Neptune[4]. If we accept machine-supported links as the essential feature of hypertext systems[5], then we can claim that SPMS is a hypertext framework for managing software.

We use the RCS Revision Control System[6], the *mkmf* makefile editor[1], the *make* program[7], and the *ninstall* software installation utility[8] in combination with SPMS to automate many of our routine software management tasks. Using a tool such as SPMS convinces us that a hypertext network of links can be effective in coordinating the use of more specialized software management tools for systems like Chipbuster.

## The System that We Manage

Chipbuster is a UNIX-based system developed by Hewlett-Packard for custom VLSI design. Key features include a high degree of configurability and integration. Functionally, it includes an artwork/schematic editor and extensive simulation and verification capabilities. A common connectivity database, common user interface and a script-based command language provide quick turnaround for IC design activities.

About 30 engineers develop and maintain Chipbuster. It contains about 600,000 lines of source code stored in 3200 files spread over 160 directories, and programmed in a variety of languages including C, Objective-C, Lisp, Pascal, Ratfor, and Fortran. The software consists of 26 programs linked with 17 libraries. When compiled with optimization, the complete Chipbuster product occupies 36 megabytes of disk space on our HP9000 Series 300 workstations. However, during development we usually compile the source code with extra debugging information and this causes our disk space requirements to grow beyond 110 megabytes for the linked programs and the object code libraries alone.

Although only 160 directories contain Chipbuster source code, the additional support directories increase the total number to over 1000. Yet, despite the large number of directories, SPMS automates the software management process so that we do not need anyone to manage the system except at the time that we do a major software release.

## Why Not Just Use Make?

The Software Project Management System is a tool for creating and traversing networks of typed symbolic links superimposed on the UNIX directory hierarchy. SPMS supports subnetworks called *projects*. Projects are groups of directories. The arrangement of the directories in a project is arbitrary but commonly organized with a number of utility directories such as `etc`, `adm`, `bin`, `lib` and `tmp`, as well as source directories such as `src` and `include`. A project may contain subdirectories which in turn may be organized as subprojects. Figure 1 shows how the directories might be organized in a simple project with one subproject.

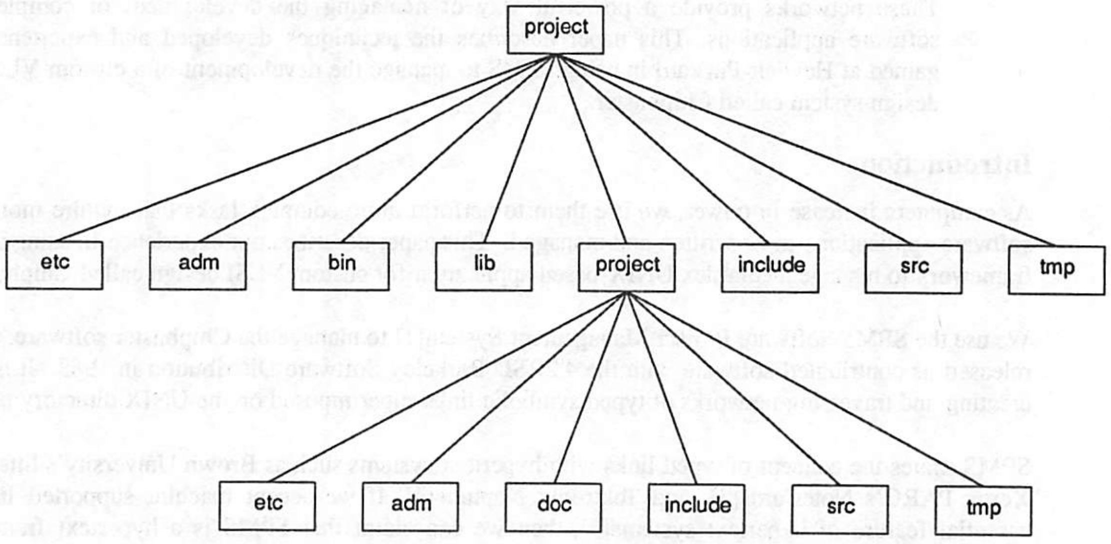


**Figure 1.** A Typical Arrangement of Project Directories

The SPMS *pexec* command provides the mechanism for executing another command in the directories belonging to a project. For example, the command

```
pexec ls
```

executes the *ls* command to list the files in each of the directories shown in figure 1. In this example, *pexec* behaves very much like the *find* command as it recursively descends the directory hierarchy. The same task could also have been achieved by using the following command from the top directory:

```
find . -type d -print -exec ls {} \;
```

While *find* does this simple job well, more complex tasks do not always need to be performed in every directory. For example, when using the *make* command to build programs and libraries, only the directories containing source code need be addressed. A common practice is to have *make* call itself recursively on each source code directory. For example, In the top directory of figure 1, we might see a makefile which contains the following commands for compiling, installing and cleaning up software contained in the `src` and `project/src` subdirectories.

```
DIRS = src project/src

all:
  for i in $(DIRS); do (cd $$i; make); done

install:
  for i in $(DIRS); do (cd $$i; make install); done

clean:
  for i in $(DIRS); do (cd $$i; make clean); done
```

Although this scheme does require that the subdirectory pathnames be stored in the makefile, this may not be a problem, especially if the directory hierarchy does not change frequently. However, each time we need to add more commands or select a different set of subdirectories based on the context of the task, we are forced to modify every makefile. This can become a major burden if there are many makefiles.

SPMS offers typed links as an alternative to the recursive *make* scheme. Each project has a file (called the *project link file*) which records the pathnames and types of the directories belonging to the project. Directories may have more than one type, and types may be prioritized. The directories in figure 1 might have the following set of directory links.

| Pathname | Link | |
|---|---|---|
| | Type | Priority |
| ./. | project | 2 |
| ./src | src | 2 |
| ./include | include | 0 |
| ./project | project | 1 |
| ./project/src | src | 1 |
| ./project/include | include | 0 |

Given this set of links, we can replace the recursive *make* command with the command:

```
pexec -Tsrc make
```

which scans the project link files for directories with the `src` link type and executes the *make* command in each directory in the sequence established previously by the priority of the link type. (Directories with the same priority are sorted alphabetically by their pathnames.)

The link type priority can also control the directory selection process. We could use the following command if we wanted to execute *make* in only source directories with priority 1.

```
pexec -Tsrc.1 make
```

In the last two examples, the *pexec* command has searched for source directories in all of the project link files and sorted the links before executing the *make* command. This method requires the priority of every link type to be chosen in the context of the entire project hierarchy, and this can require considerable effort if there are a large number of links or link types. In some cases we can reduce this effort by just assigning priorities to each project and executing a non-recursive *pexec* command in each project to select the appropriate directories. The following command runs *make* in the same sequence as before.

```
pexec -Tproject pexec -r -Tsrc make
```

Finally, we need some way of avoid having to create new link types for every task. SPMS allows link types to be combined in boolean AND, OR, and NOT expressions. Suppose that we wanted to search for the defined constant BUFSIZ in all of our source and included files. By combining the `include` and `src` link types with a logical OR operator (and quoting it to protect it from the shell), we can search the files in both the source and included file directories:

```
pexec -T'include|src' 'grep BUFSIZ *.h *.c'
```

The network of typed links supported by SPMS provides a powerful mechanism for managing large software projects, yet costs very little in terms of disk space. The size of a project link file rarely exceeds 3K bytes, and in a network of 100 projects the average size of a project link file is typically less than 1K.

## How Do We Manage Our Software?

The Chipbuster software management system provides support for version control, building and debugging, module and system testing, release, manufacturing, distribution, and subsequent fixes to the software. The flow through each of these phases is shown in figure 2.

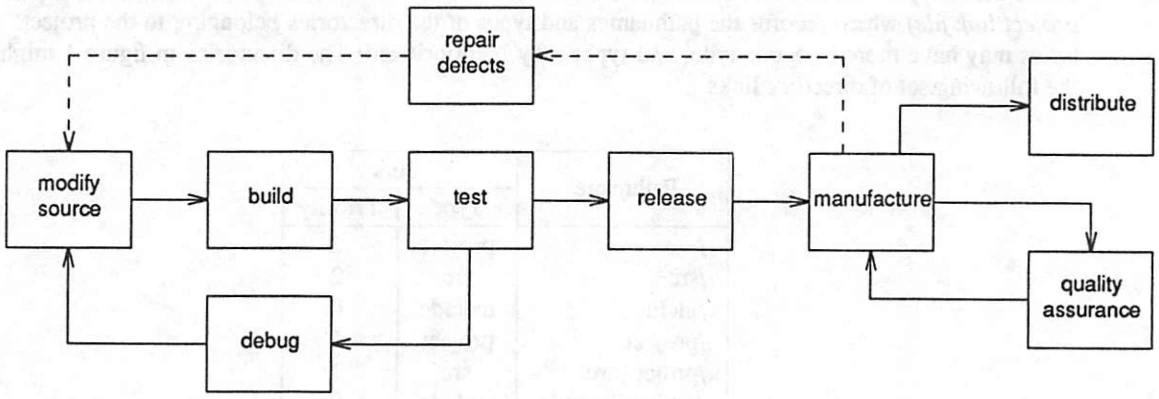


**Figure 2.** Chipbuster Software Management Process

### *Project Structure*

We manage the Chipbuster system as a three-level hierarchy of SPMS projects with one library or program per project. Each project consists of a set of support directories plus a number of subproject directories. Figure 3 represents a (small) portion of this directory arrangement with 3 programs (trantor, guide, and terminus) for editing, simulation, and IC module generation respectively, and two libraries (periphery and stacks) that provide common command parsing and database management functions. Trantor itself consists of several subprojects of which the editor project is but one.

### *Version Control*

The master source for Chipbuster is maintained using the RCS Revision Control System. Each project has at least one source directory, each with its own subdirectory containing the RCS version control files. Because the source directories are used as the working area for nightly compilation of the system, a working copy of the latest version of each source file is always kept checked out in its own source directory.

It is rare for anyone to modify the master source on the central source code control machine; all development is carried out on individual engineer's workstations. Instead, we provide utilities that perform remote check-in and check-out using network file access. These utilities invoke the RCS *ci* and *co* commands after setting up the network connection. In addition to updating the master source, the remote check-in utility (called *cbci*) performs one additional step by checking out a working copy of the file on the master source machine.

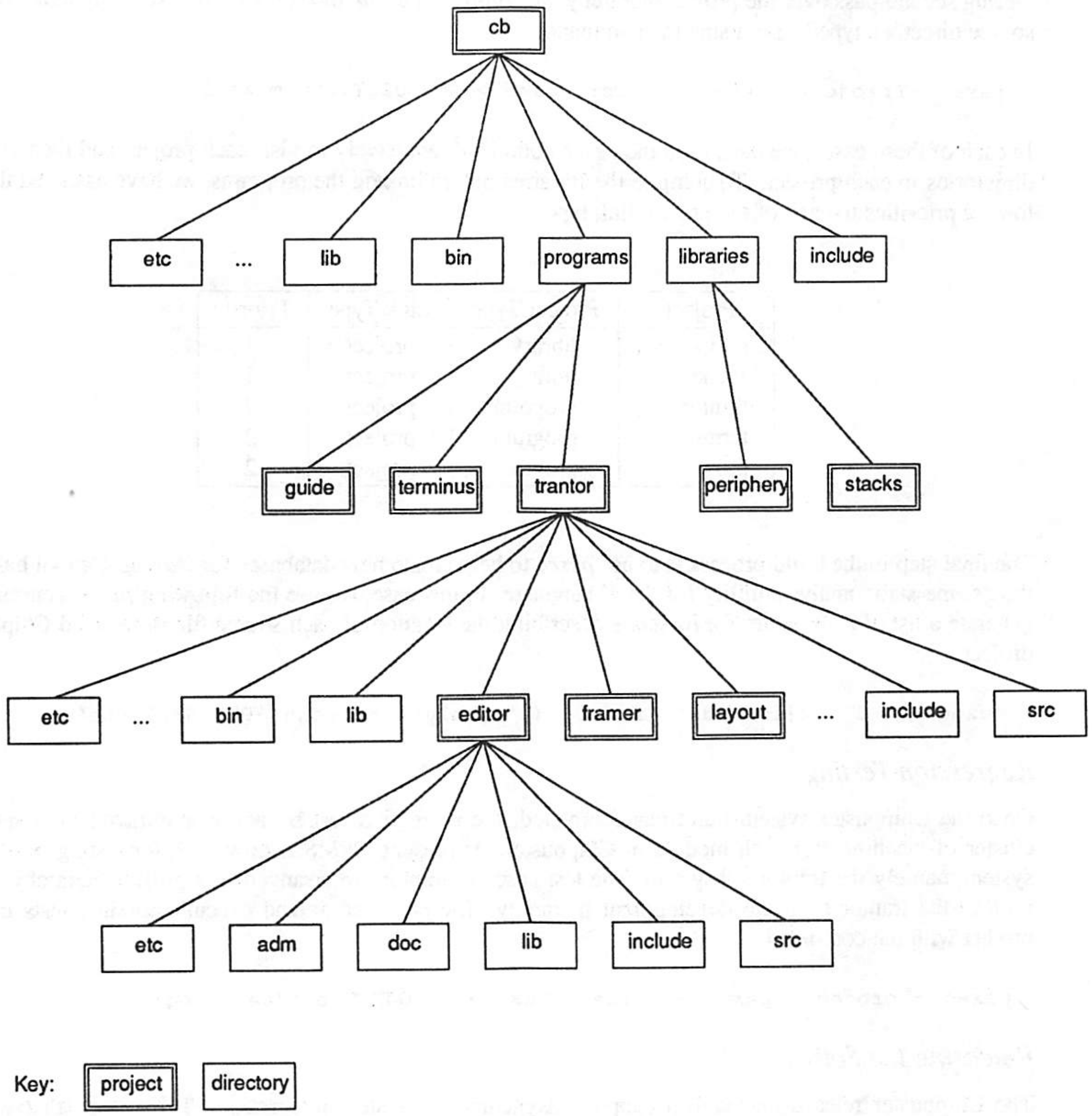


**Figure 3.** Chipbuster Software Directory Hierarchy

A common software configuration management issue, and one that has been solved many times[9,10], is to record the versions of the source files in each software release. We use RCS symbolic names for this purpose and mark each program and library with a unique version number related to these symbolic names.

## *Building the System*

The Chipbuster software is built nightly under the control of the *cron* program. The first step of the process is to sweep through the entire project hierarchy and recreate makefiles for the *make* program from predefined templates using the *mkmf* makefile editor in the following command:

```
pexec -Tproject pexec -r -T'libmake|binmake' 'mkmf > $PROJECT/etc/mkmf.log'
```

Here we see a boolean expression, `libmake|binmake` selecting a combination of directories when the *pexec* command executes the *mkmf* command in each project directory typed `libmake` or `binmake` to create library makefiles or program makefiles respectively.

During second pass over the project hierarchy we compile and link the software by executing *make* in each source directory typed `src`, using the command:

```
pexec -Tproject 'pexec -r -Tsrc make > $PROJECT/etc/make.log'
```

In each of these cases, we have used the *pexec* command recursively to visit each project and then visit the directories in each project. To compile the libraries before linking the programs, we have assigned the following priorities to each of the project link types:

| Project | Project Type | Link Type | Priority |
|---|---|---|---|
| periphery | library | project | 1 |
| stacks | library | project | 1 |
| trantor | program | project | 2 |
| terminus | program | project | 2 |
| guide | program | project | 2 |

The final step in the build process is to use *pexec* to help create new databases for *ivoscope*, a tool based on the *cscope* static analysis utility for the C language. In this case, we use the following *pexec* command to generate a list of path names for *ivoscope* describing the location of each source file in selected Chipbuster projects.

```
pexec -q -T'src|include' 'for i in *.[chly]; do echo $PWD/$i; done'
```

### *Regression Testing*

Once the Chipbuster system build has completed, the regression test sequence is initiated on a separate cluster of machines for each module of Chipbuster. At present, SPMS is only used for testing part of the system, namely the trantor subsystem. The test cases in trantor are arranged in a project hierarchy which mimics the trantor software development hierarchy. The *pexec* command executes module tests in each project with the command:

```
pexec -Tproject 'pexec -r -Ttest Test > $PROJECT/etc/test.log'
```

### *Releasing the Software*

The Chipbuster release mechanism supports asynchronous system integration. This means that we can freeze a version of a program or library, copy it to a location where is can be shared by other software developers, and then proceed on with development without having to wait for other developers to release their projects.

The basic release process for each project is controlled by the *Release* command which calls on *pexec* several times to perform the following steps:

1. Change the permissions on the RCS version control directories to prevent changes to the source code during the release process.
2. Increment the version number of the program or library. Usually, this means incrementing the version number of a file called `versiontbl.c` which is later compiled with the program or library.
3. Rebuild the makefiles using the *mkmf* command.
4. Recompile any out-of-date source code with the *make* command.
5. Copy programs, libraries, scripts, and source code to the parent project.

6. Mark the source code with RCS symbolic names derived from the incremented version number of the `versiontbl.c` file.

7. Restore the permissions of the RCS directories.

Figure 4 shows the movement of the software up the directory hierarchy as the editor subproject is released, and then the trantor subproject. Object libraries and programs are copied to directories where they can be shared. Source files are stored temporarily in the manufacturing (`mfg`) directory hierarchy of each project, in preparation for the final release of the complete system. The release process finishes when all of the subprojects have been released to the top level and the entire system passes all of the regression tests.

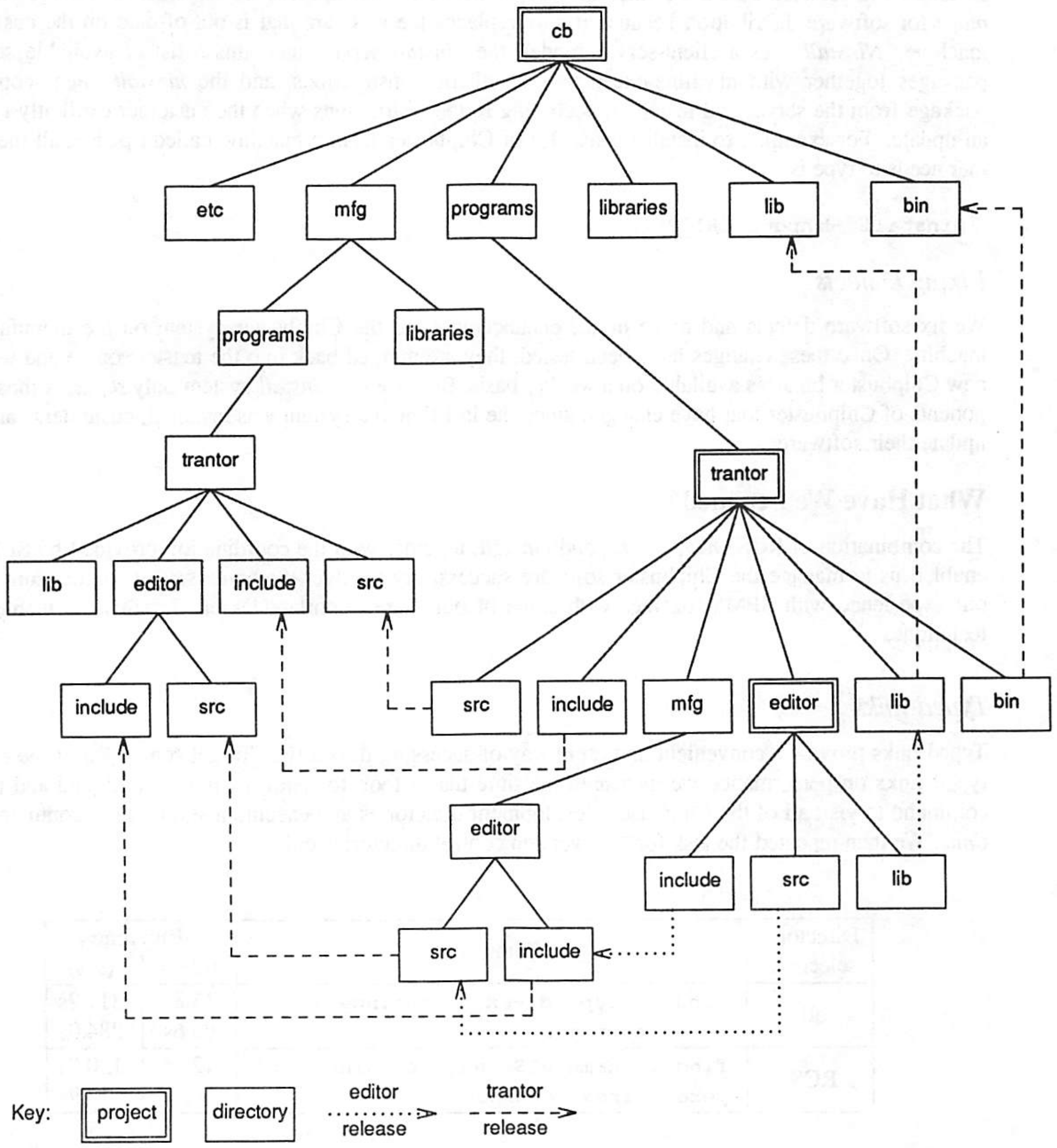


**Figure 4.** Chipbuster Release Process

### *Manufacturing and Distribution*

On completion of a Chipbuster release, all of the released source code and software management infrastructure is reconstituted on the manufacturing machine with the same directory structure as the Chipbuster development system in preparation for final product assembly and user testing. The only difference is that we symbolically link all of the `bin` and `lib` directories in the project hierarchy to one top-level `bin` and `lib` directory respectively. This avoids having to rerelease projects since we are already dealing with software that has been released from the development system.

After we assemble the Chipbuster product, we use a tool called *ninstall* to distribute the product to other divisions via Hewlett-Packard's internal TCP/IP network. *Ninstall* can be considered the equivalent of *make* for software distribution because it only replaces the software that is out-of-date on the customer's machine. *Ninstall* uses a client-server model: the *ninstall* server maintains a list of available software packages together with machine-executable installation instructions, and the *ninstall* client copies the package from the server and installs it according to the instructions when the customer explicitly requests an update. For example, to install release 1.2 of Chipbuster from a machine called `hpdtc`, all the customer needs to type is

```
ninstall -hhpdtc CB1_2
```

### *Fixing Defects*

We fix software defects and make minor enhancements to the Chipbuster system on the manufacturing machine. Once these changes have been tested, they are merged back into the master source and we make new Chipbuster binaries available on a weekly basis. Because the *ninstall* system only replaces those components of Chipbuster that have changed since the last time the system was installed, customers can easily update their software.

## What Have We Learned?

The combination of RCS, *mkmf*, *make*, and *ninstall*, together with the coordination provided by SPMS has enabled us to manage the Chipbuster software successfully for over three years. This section summarizes our experience with SPMS together with some of our more important lessons in software management techniques.

### *Typed links*

Typed links provide a convenient and rapid way of accessing directories. To get some idea of the effect of typed links on performance we measured the time that it took for both the *pexec* command and the *find* command to visit all of the Chipbuster development directories and execute a simple shell command, *continue*. We then repeated the task for `RCS` version control directories only.

| Directory selection | Command | Run time (user) | Run time (sys) |
|---|---|---|---|
| all | `find . -type d -exec continue \;` | 13.8s | 317.2s |
| | `pexec continue` | 63.6s | 284.0s |
| RCS | `find . -name RCS -exec continue \;` | 12.3s | 130.0s |
| | `pexec -Trcs continue` | 6.0s | 17.9s |

*Pexec* is slower than *find* in the case where every Chipbuster directory is accessed, but is over six times faster when accessing just the version control directories. However, notwithstanding the improvement in performance, the real advantage of typed links is in providing a convenient way of selecting directories for software management. If there is any disadvantage, it is in the effort required to keep track of the various link types and where they are used, especially as the network grows. We found that we had to develop guidelines and prototype project link files to help engineers set up consistent projects.

## Version Control

A significant drawback to the way that we currently organize our source code is that we keep the working copy of the latest development source code in the same area as the master version control files. We adopted this scheme because RCS expects to find version control files in the same directory as the source files, or in a subdirectory called RCS. However, we have found that this arrangement prevents us from checking out more than one version of the source at a time. If we want to retrieve an earlier release, we have to clobber the most recent changes to the working source, and hence interfere with the nightly build of Chipbuster. We need to keep the working source files separate from the master version control files.

RCS symbolic names do not provide enough support for configuration management. We should have used parts lists because they can be copied to the manufacturing machine along with the rest of the source code, and updated when defects are repaired.

## Building the System

Being able to assign priorities to typed links has made it easy for us to select and control the order in which we build Chipbuster programs and libraries. Each time we add another subsystem to Chipbuster, we can give the new project a priority that will enable the software to be built in the correct sequence.

In the past, we have experimented with various alternatives to the traditional *make* program. However, we have found that a key issue in maintaining a large system such as Chipbuster is not having to keep makefiles at all, but rather use the *mkmf* utility to create them just prior to running the *make* program.

We have run into difficulties when using the same copy of the source code to compile Chipbuster for different machine architectures, for debugging, profiling, or optimization. Unfortunately, the classic *make* program does not allow us to separate the source and object code in different directories. We need to migrate to another utility such as *mk*[11] which will support this objective.

## Regression Testing

As the Chipbuster regression test suite grows, we are finding it harder to complete all of the module tests before the development activities begin the next morning. Only our system tests run in parallel on clusters of machines and we need to look at similar mechanisms for running the individual module tests in parallel. Although we could use SPMS link priorities to help us decide what can be run in parallel, this is not the best solution. Since most, if not all, of the module tests are independent of one another, we could to use a load-averaging scheme to spawn tests on lightly loaded machines mounted via NFS to the central test hub machine.

## Releasing the Software

The release process has been effective during ongoing development - individual engineers can release their projects to their parent projects just by invoking the *Release* command and then continue with their work, unhindered by the release schedules of other projects.

However, we discovered that this mechanism does not work as well as we had hoped when it comes time to do a major Chipbuster release. We discovered that we nearly always had to backtrack and release library subprojects several times to correct unanticipated problems in the interfaces between programs that only became apparent during system testing. This in turn would perturb the program depending on those libraries which would then need to be rereleased. The only way that we could carry out a major product release successfully was to freeze all changes to the master source and have one person release the entire project hierarchy with the command

```
pexec -Tproject Release > $PROJECT/etc/Release.log
```

until the release was completed and the system was transferred to the distribution machine. Even though the release process takes only about 4 hours for the whole Chipbuster system, we find that we usually spend about a week to finish a major release.

The release process was an experiment. We built too much knowledge into the *Release* program about file suffixes. Each time we had a new type of file to release, we had to update the program. This information really belongs in the individual makefiles.We learned that copying the source code to the `mfg` directory hierarchy is not a good idea in terms of disk space, especially when there more than one major release in progress at the same time. Versioned source code parts lists would have been more effective and less demanding space-wise.

### *Manufacturing and Distribution*

Since we adopt the same software management scheme for building Chipbuster on the manufacturing machine as we do on the master source machine, we find it easy to build each Chipbuster product release and have it available for our customers to install over the network. Even though the size of the product is about 36 megabytes, we find that network distribution over 56 kilobit connections has obsoleted physical tape distribution methods.

### *Fixing Defects*

We have had very favorable results with our scheme for fixing defects and updating the software without having to endure the upheaval of a major Chipbuster release each time. Since the changes are usually requested by our customers, they are willing to accept frequent updates via the *ninstall* distribution mechanism.

We would like you to believe that we have mastered the management of multiple releases. However, we still struggle with tracking the changes that have been made to the released version of the Chipbuster system. It is relatively simple matter to merge modifications back into the master source. However, we do not have a scheme for updating the RCS symbolic names for each minor release. At present we only stamp each source file with a symbolic name at the time of a major Chipbuster release, and keep a separate record of the names and versions of the source files that have changed since that time.

## Future Plans

Our experience with managing the Chipbuster system makes it clear that we have several areas where we can make improvements in our software management techniques and in the realm of hypertext frameworks.

We need to separate the source code and object code files into different directories so that we can compile Chipbuster in different contexts or for different machine architectures from one copy of the source code. We must separate the working source code from the version control files so that we can check out multiple versions of the system. We need to do a better job of recording the versions of the source files, object code libraries, and programs that make up each release so that we can reconstruct previous releases more easily. It would also be helpful to provide some way of documenting the changes to the source so that we can trace all of the changes that have made to the system between one release and the next.

Given our experience with SPMS, we see how a hypertext network of links can be used to coordinate more specialized software management tools. We will reuse the idea of a system of typed links together with the following features for our next framework:

- Interfaces for program access, batch processing, and graphical views of the network.
- A graphical monitor to show the progress of a *pexec* command as it traverses the project hierarchy.
- Relational link type expressions for more precise directory selection.
- Parallel execution of commands based on link type priorities.
- Easier maintenance of the link types and their priorities.
- A way of generating link type priorities based on the dependencies of programs on libraries.

## Conclusions

The combination of the SPMS Software Project Management System, the *mkmf* makefile editor, the *make* program, and the *ninstall* network installation utility has helped us automate many of our routine software management tasks. The network of typed links offered by SPMS provides us with a flexible infrastructure without enslaving us to one software management system. We gained a great deal of experience and perhaps a little humility as we learned how to manage the development of Chipbuster. As we continue to improve our tools, we try to remember the following principles:

1. The speed of the software management system controls the number of build-test-release iterations which in turn can decide the quality of the final software product.
2. The disk space needed for software development inevitably exceeds that which is available - carefully choose what software management information is stored, and recreate anything else when needed.
3. Hypertext and object database management systems may not scale up unless they support partitioning of subsystems. Use nested systems and try to manage objects of similar granularity in each system.
4. More powerful software management tools lead to more complex software, not easier software management.

## Acknowledgements

In 1985, SPMS was ported from 4.2BSD to HP-UX, Hewlett-Packard's version of System V UNIX operating system. Soon after, it was adopted as the foundation for the Chipbuster software management system. This system was developed over a three year period by Fred Harder, Martin Gates, Dell Lee, and David Pinsky. It has seen active use throughout this time. Bill Ward is the most avid user; he is the engineer in charge of manufacturing the Chipbuster product.

## References

[1] Peter J. Nicklin, The SPMS Software Project Management System, *4.2BSD User Contributed Software Supplemental Manual*, University of California, Berkeley, August 1983.

[2] L. Nancy Garrett, Karen E. Smith, and Norman Meyrowitz, Intermedia: Issues, Strategies, and Tactics in the Design of a Hypermedia Document System, *Proc. of the Conference on Computer-Supported Cooperative Work*, MCC Software Technology Program, Austin, Texas, December 3-5, 1986, pp. 163-174.

[3] F.G. Halasz, T.P. Moran, and R.H. Trigg, NoteCards in a Nutshell, *Proc. CHI+GI '87 Human factors in Computing Systems and Graphics Interface*, SIGCHI Bulletin, April 1987, pp 45-52.

[4] Norman DeLisle and Mayer Schwartz, Neptune: A Hypertext System for CAD Applications, *Proc. of ACM SIGMOD '86 International Conference on Management of Data*, Washington, D.C., May 28-30, 1986, pp. 132-143.

[5] J. Conklin, Hypertext: An Introduction and Survey, *Computer*, 20(9):17-41, September 1987.

[6] Walter F. Tichy, RCS - A System for Version Control, *Software Practice and Experience*, 15(7):637-654, July 1985.

[7] Stuart I. Feldman, MAKE - A Program for Maintaining Computer Programs, *Software - Practice and Experience*, 9(4):255-265, April 1979.

[8] Tai Jin, Norm Kincl, Brent Thompson, and Charles Untulis, Ninstall - Network-Based Software Distribution, *UniForum '88 Proceedings*, Dallas Texas, February 1988.

[9] Eugene Cristofor, T.A. Wendt, and B.C. Wonsiewicz, Source Code + Tools = Stable Systems, *Proc. of the Fourth Int'l Computer Software and Applications Conference*, Chicago Illinois, 1980, pp. 527-532.

[10] David B. Leblang and Robert P. Chase Jr, Computer-Aided Software Engineering in a Distributed Workstation Environment, *ACM SIGPLAN Notices*, 19(5):104-112, April 1984.

[11] Andrew Hume, Mk: a successor to *make*, *USENIX Conference Proceedings*, Phoenix Arizona, June 8-12, 1987, pp. 445-457.

# Under 10 Flags

## (not always smooth sailing)

*David Tilbrook*

Nixdorf Computer Canada Ltd.

*ABSTRACT*

**Creating and managing source that can be installed successfully across a wide range of systems and machines is not easy!**

**This paper discusses some of the issues and problems that arise when managing a moderately large body of software on a wide variety of machines and environments.**

**The paper briefly presents the general approach, some of the strategies used to overcome the many differences between systems, and finally registers some complaints and challenges to the reader.**

**The objective in writing this paper is to provide background information for the closing presentation at the 1989 Usenix Software Management Workshop. The author and Barry Shein of Encore are the co-chairs for this workshop.**

## 1. Introduction

Despite my use of UNIX† since 1975, I cannot really call myself a UNIX user. The system I use is the sixth edition (circa 1977), plus a source control system from PWB/Unix (the SCCS system), the SIGSTOP signal from [24].?bsd, five of the settings provided by termcap or terminfo databases (I don't care which), the DBM database of the seventh edition, a small subset of *csh*(1) (modified to incorporate my cursor line editor), and some networking. However, I do supplement this somewhat modest subset with 250 other programs that I refer to as the D-tree.

I am quite satisfied with this approach to UNIX, but I appreciate that the reader might think that this is due to the lack of alternatives.

This is not the case.

At this very moment I have a choice of five different flavours of UNIX and at least four (maybe more) window managers. As I write this document, there are seven graphic workstations within hitting distance, all unoccupied. I have made several serious attempts to use publicly available window managers, but, each time, I decided that I could not afford the aggravation and the considerable loss of performance and effectiveness.

But this paper is not to proselytize my approach to UNIX. It is to discuss how I deal with a problem that I have, given that approach.

† UNIX is a trademark of Bell Laboratories.

That problem is that I need to install the D-tree on every machine that I use or plan to use, and porting 8 megabytes of source (including documentation) is not as easy as you might think. There are little differences among the various flavours of UNIX that introduce some minor[1] impediments.

But I am relatively confident when I state that I have evolved a strategy and its supporting tools that can be applied to create and install software on virtually any UNIX system[2].

In support of this claim, during a four hour meeting on Jan 31st, 1989, I installed the D-tree on three different environments. This required creating a 30 line configuration file for each machine (by editing a copy of the prototype) the fixing of a sign extension problem reported to me by another user, and the modification of four other sections of code.

All of the latter modifications were required to compensate for the host's non-conformance with the base system (e.g., 4.{1,2,3}bsd unix5.{1,2,3}) and could be attributed to the schizophrenia of dichotomous[3] UNIX systems. (An example of one of these problems will be given later.) All of the constructions and installations used the same source files, via a distributed file system, without copying, linking, or changing the kernel's *namei* routine.

However, for me to claim success based solely on my experience, should be considered in the same manner as one would consider an Athena project leader saying he had no problem installing X11.

There are a number of other testimonials[4] including one from an relatively inexperienced D-tree user (Nick Kosche at Sybase Inc.), who performed multiple first-time installations in three days, with some phone consultation with Pat Place of the SEI. One of his installations was on a new (for the D-tree) machine, a new operating system, and a new version of C[5] yet the installation gave rise to only three previously unencountered problems: token pasting, modification of constant strings such as *mktemp*("/tmp/xXXXXXX"); and C compliance over entire files. (For example, even the conditionally suppressed sections have to be tokenized.) Pat has conveyed to me that Nick has very a positive attitude towards the experience. I find that encouraging.

## 1.1. Warning - reading the following may be upsetting

One of the recently encountered problems merits presentation due to its being a particularly good example of the kind of problem one faces in trying to make portable code:

On a system, that shall remain nameless (to protect the guilty), the header file <sys/stat.h> contained

```
        #define S_IFIFO S_IFLOCK /* for sys5.3 compatibility */
```

Unfortunately one of my programs, which had been successfully ported to some 35 different flavours of Unix, contained:

```
        #include <sys/stat.h>
        ...
                switch (...) {
        #       ifdef S_IFIFO
```

[1] "minor" when compared to the English Channel.

[2] The D-tree strategy and toolset is being used on non-UNIX systems, but not by me, so I cannot eliminate the qualification.

[3] Very close to "dicey" in the dictionary.

[4] provided on request

[5] ANSI C is a dialect of C, just as English is a dialect of German.

```
        case S_IFIFO:
        ...
#       endif

#       ifdef S_IFSOCK
        case S_IFSOCK:
        ...
#       endif
        }
```

which raises the extremely useful and informative

```
duplicate case label 49152 in switch
```

diagnostic. The solution is rebarbative and not foolproof (never underestimate the ingenuity of a fool) but amusing.

```
/* On some systems S_IFIFO is defined as S_IFSOCK
**      so undefine it
** Let us hope that no one ever
**      defines S_IFSOCK as S_IFIFO!
*/

#if defined(S_IFIFO) && defined(S_IFSOCK)
#       if (S_IFIFO == S_IFSOCK)

#               undef S_IFIFO

#       endif
#endif
```

Problems such as the above are exasperatingly common. In the following sections, I will try to explain some of the D-tree philosophy (and the approaches employed) to ensure that installation continues to be relatively painless.

However, the philosophy does depend on the availability of tools beyond those offered by vanilla UNIX. To describe the construction suite in full is beyond the scope of this paper. There is a brief outline of some of the major goals for the system in the next section, followed by a listing of some of the more important aspects of the approach.

## 2. Quod <EratlEstlErit> Faciendum (qef)

*qef* is the most recent of a series of my UNIX construction systems, starting with work done with Tom Duff at the University of Toronto in 1976. *qef* (a.k.a., *qed++*) is primarily a driver used to control software construction and installation. Although *qef* has evolved and changed dramatically as it has been applied to more and more applications and systems, the primary objective of the research remains unchanged, and this was, and is, the creation of an approach to software construction and installation that meets the following criteria:

### Constructions and Installation without Change to Source

The construction system should support the initial installations of large collections of software, on a variety of machines, at a variety of sites, without requiring the modification of **any** source, using a trivial configuration mechanism (e.g., fill in the blanks) and a single command (e.g., "qef Instal Man Post[6]").

[6] The *qef* systems permits the division of installation into three phases: software installation; installa-

*source* is defined to be **all** the information that is created manually or must be provided by the supplier. This, by definition, includes **all** the information used to control construction and installation, such as *make*(1) files.

**Ease of Distribution Upgrade or Modification**

The system should support the upgrading of a previously installed body of software by the simple addition of new source files and/or the removal or replacement of old source files, and the issuing of a single command (e.g., "qef Instal"). Furthermore, the addition or removal of source files should rarely, if ever, require the modification of the construction control files.

**Controlling Information Specification and Use**

The system should provide mechanisms to specify and/or retrieve required control information (e.g., the target location of the installation) from one and only one place, or through one and only one interface. In other words, user supplied information should never have to be expressed more than once. A user should be able to feel confident that all aspects of the construction system that need such information, should retrieve it in a manner that ensures consistency across the entire system and all uses. Furthermore, it should be possible to ensure that correct and consistent values are being used no matter what part of the system is invoked. That is to say, the information should be the same when retrieved during a full or partial construction, or the invocation of some subset of the construction system.

An extension, or almost a prerequisite, of this objective is that the system should support the trivial addition of new pieces of construction control information. For example, if the system needs to support *widget* compilation, there should be a simple mechanism to specify the name of the *widget* processor and its normal flags through a unique and universally accessible interface.

Finally, if the default value of a piece of control information needs to be changed or overridden within some subset of a source distribution, it must be ensured that the modification is effective over the entire subset.

**Reduce Required User Supplied Information to Minimum**

The system has to provide mechanisms to express construction specifications as succinctly as possible.

For example, in a directory of source that is compiled and archived into an object library, it should be sufficient to state: "build and install a library called X". >From this statement alone, the system should be able to generate all the required information that will perform that task.

The mechanism used to convert simple specifications into the full construction procedures must be applicable to as wide a range of applications as possible. It should be trivial for a user to add new procedures[7]. Furthermore, it must be possible to state possible variations to a procedure easily and tersely. Such variations should not require the total restatement of the construction rules[8].

tion of documentation; and post installation procedures.

[7] In *make*(1) one can add new suffix rules to an individual *make* file, however, that rule must be added to every file that might use it. To make it universally available normally requires recompiling the *make* program itself.

[8] Traditional *make*(1) provides suffix rules that are used to specify the default constructions for an object. But, if any variation from the normal rule is required, the entire construction, incorporating that variation has to be specified.

**Phased Construction and Distribution Subsets**

The construction system must support partial or phased constructions such that they are absolutely equivalent to the same constructions taking place as part of a full construction. That is, the user should be able select parts or phases of the construction to be done and be assured that the behaviour is equivalent to those selected phases being done as part of a full construction.

Furthermore, there should be few (if any) modifications required if the source to be processed is a subset of the full distribution. For example, one should not have to modify the construction control information when parts of the full distribution are excluded for economic, licensing, or other reasons.

**Consistency Between Incremental and Full Constructions**

The system must guarantee the consistency of objects produced by an incremental construction with those that would be produced by a full reconstruction. Incremental reconstructions are defined to be those constructions that skip some construction steps if the object to be produced is determined to be up-to-date. The construction system should ensure that an object is reconstructed if there is any aspect of its construction that has changed, that might result in a significant change in the object itself.

**Support for Testing and Development**

Finally, the system should facilitate testing and modification without endangering the production versions, whilst ensuring the almost trivial addition of those changes or extensions when completed. Experimentation with the source by a programmer should be easy to initiate and economic (i.e., not require large amounts of disk space). The system should ensure that the behaviour of a test system in a non-production environment is truly reflective of the behaviour of that system in the production environment.

## 2.1. Caveat

The above objectives are ambitious. The task of proper system construction and installation is a difficult one. Any approach that attempts to solve the problems of large scale software will itself be complicated and/or expensive. It will have to deal with a great many special cases (the ability of programmers to create new ways of complicating the lives of the software manager is unbounded).

The *qef* system is a tool-based approach. Instead of depending on a single tool (e.g., *make*(1)) and the willingness of people to spend their time (wasting their employer's money) manually creating *Makefiles*[9], *qef* depends on a disciplined approach to the organization of source trees, some adherence to naming conventions, and the avoidance of wildly divergent construction techniques within a single directory. If these criteria are met, then *qef* can apply its tool set to build far more accurate, flexible, and efficient constructions that can be done manually, at a huge saving time and effort. Decreases in the amount of manually generated information of three orders of magnitude are not uncommon, and as often as not, the conversion finds many errors in the original information.

But, one does have to learn about some new tools and languages and adopt disciplines that might not conform with one's usual approach.

[9] The traditional use of 1000 line *make* scripts is not only time wasting, but the scripts themselves are inevitably incomplete, inflexible and, invariably, poorly maintained and wrong.

## 3. Parenthood

I have been fortunate that there have been a number of people[10] that have been willing to learn the tools and in turn made considerable contributions. Given Pat's continued participation and enthusiasm, as part of the preparation for this paper I asked him to help to enumerate the major aspects of the approach/philosophy. The following is that enumeration.

### 3.1. Constantly worry about portability

We have made a habit of assuring that any new code or modifications are tested on as many different environments as soon as is convenient. When a portability problem is discovered, a thorough search of all the source is done for any other occurrances of that problem immediately. Furthermore, if the problem can be avoided by adopting a stylistic change, we do so. Occasionally we will recheck the source for introduction of new instances of past problems.

One recent pay off for this approach was the ease of conversion to ANSI C. We were informed of the problem of having comments on "#else" or "#endif" statements some years ago and immediately found and fixed all instances, even though it was not necessary until last month.

### 3.2. Frequent testing of the testing process

In addition to the testing of the programs, the construction process itself must be exercised extensively. Complete reconstructions must be attempted on a regular basis. Any failure to complete necessitates the test be repeated in its totality until it succeeds. (Unfortunately this requirement has to be relaxed on those systems where the mean time between file system failures is measured using the hour hand instead of a calendar).

### 3.3. Adopt a common coding style

Enforcing a common coding style inevitably results in long and bitter arguments, but, when one considers maintaining 2000 source files in co-operation with others, being able to confidently apply rules for searching and processing the text has substantial returns.

For example, all routines in D-tree C source files have the following form:

```
<type>
name(arguments, ...)
    type arguments;
    ...
{
    ...
}
```

Consequently:

```
grep '^[a-zA-Z0-9_]+(' files
```

will find all routines in the system, and nothing but the routines. Going back one line gets the type. Going forward to the first line matching '^{' captures all the arguments.

Another simple but defendable rule is never using multiple declarations as in:

[10] Calvin Delbarre, Robert Biddle, Adrian Pepper, Derek Thorsland and Mark Brader, all formerly of BNR, Andy Greener of Systems Designer Ltd. and Imperial Software Technology, Jim Oldroyd of the Instruction Set, Pat Place and Mike Day of IST and now the SEI, and Tom Lord of CMU, to list a few.

```
int i, j, k;
char *p, c, **p, carray[];
```

as doing so prevents one from easily changing the type for one of the variables and not the other.

## 3.4. Always All, Everywhere, Always, or Never - Never Sometimes

We attempt to ensure that special cases are avoided as much as possible. For example:

- for ALL D-tree library source directories, ALL the files that have the same suffix, are treated in exactly the same way.
- ALL variables routines in libraries that can be static, are static.
- ALL source files have a suffix or a specific name that is ALWAYS spelled in the same way (e.g., always "README", never "ReadMe" or "Read_Me").
- *cc*(1) "-D" flags are NEVER required. Any options or configuration parameters are ALWAYS put into header files.
- For EVERY object file (other than links) there is a deterministic method for finding its source or its source location. For example, for any installed library "libX.a" there is a directory called "libX"; for any "file.o" there is a source file called "file.X". For every installed file "X", there is a file in the source tree called "X" or "X.Y", or "X" is a link to file "Z" for which this rule applies.
- ALL library routines contain an SCCS line of the following form:

  ```
  #if    !defined(lint) && !defined(NOSCCS)
  char   SIDfilename[] = {"%Z%%Q%(%M%)<tab>%I% - %E%"};
  #endif         /* not lint or NOSCCS */
  ```

  The "%Q%" is ALWAYS "-l<libname>".

This way:

```
wot -o binaries
```

ALWAYS produces output that can be processed using simple stream editors to generate the list of files that were used to produce the program.

- ALWAYS use:

  ```
  #include   <file.h>
  ```

  NEVER use:

  ```
  #include   "file.h"
  ```

  The latter form prohibits creating a modified version of "file.h" in the current directory without copying into that directory all the files that might "#include" it.

There are many other rules that could be listed, however, the above should be sufficient to illustrate the principle. However, there will ALWAYS be someone who objects to one of the above restrictions. Some people are NEVER satisfied.

## 3.5. Separation of Object and Source

The D-tree construction tools, due to contributions by Tom Lord, support the total separation of source and object files, using a source path mechanism. This is NOT done using virtual directories or links, but through full path names and regeneration of the construction script if a source file moves up or down the source path.

This does occasionally cause problems. Source path names can exceed the limits imposed by language processors (one version of *cc*(1) truncates files names in the stab entries at 50 characters). Many versions of *cc*(1) can only support eight "-I" flags which can easily be exceeded when there are three directories in the source path.

However, there are very real advantages to the scheme that are not realized using links or virtual directories. Furthermore, our approach is cheap and works on all UNIX systems in the same way.

## 3.6. Reduce your dependence on others as much as possible

It is amazing, and somewhat dismaying, how unreliable that other guy's system is. I know your system is okay, but his *basename* does not work in certain cases.

Currently, the only tools that the D-tree construction uses, that it does not provide itself, are:

> ar, as, awk, cat, cc, comm, cp, grep,
> lex, lorder, make, mkdir, mv, ranlib, rm,
> sed, sh, sort, touch, tsort, & yacc[11]

Other tools can be added to this list relatively easily, but are done so in manner that permits a unique easily overridden specification and the location of all uses using *grep*(1).

In the D-tree EVERY (boring isn't it) tool use is prefixed by "_T_". The macro is defined using:

```
#conddef _T_tool ...
```

"#conddef" is similar to "#define" but does the definition only if the macro is not already defined.

## 3.7. E2BIG - U2SMALL

Many of the D-tree tools support the facility to specify that the arguments to be processed are contained in the standard input or the argument files. The commands:

```
grep pattern `cat SourceList`

        or

xargs grep pattern <SourceList
```

fail or are unacceptably slow when *SourceList* contains thousands of files and tens of thousands of characters. Quick and efficient ways to process large lists of files are essential.

## 3.8. Cleanliness is near to Portliness

The removal or creation of a single file can be extremely damaging and can frequently invalidate any testing or construction. Tools to check the the source, object, and installed trees are an important part of the D-tree construction system, used to help manually maintain lists of the source, object and installed files. During periods of heavy activity this package is invoked on an almost hourly basis. It can be tedious, but it yielded a very high return on investment the day a file server silently misplaced 72 of my source

11 *make*(1) is employed sparingly for very simple scripts. *sed*(1) only required to deal with *yacc*(1) deficiency, namely eliminating name clashes. *awk* is used once in the bootstrap to perform a relatively simple task that should probably be done using *ed*(1), since: "If you have a problem and you think *awk*(1) is the solution, then you have two problems." - David Tilbrook

administration files, along with a number of other files that could be reproduced. The file list package discovered the loss within an hour and provided all the information I required to create the current version of the software. (Fortunately there was no file for which both the 'g-' and 's-files' had been lost.)

Unfortunately, I was unable to reproduce the system as it had existed the previous day (thus preventing any regression testing) as that was also the day that it was discovered that the backup system had failed for the previous 5 nights.

## 4. Configuration and lets play "Hunt the Header"

This is the penultimate section, and as such is probably the last opportunity to present something concrete. In previous sections I referred to the configuration file and the unique source of all system dependent information.

The only difference between two D-tree installations is the specification of the two arguments to the SetUp command. The "-x" flag to SetUp yields:

```
SetUp [ -x ] [ -s SrcDir ] Machine_file Object_dir

initial setup of a dtree distribution

-x            Display these explanations
-s SrcDir     name to be used for source directory
machine_file  name of the machine file to be used
Object_dir    directory to contain objects (default ..)

Shell script builds the required files to in Object_dir/Magic to
build the Dtree system. The machine file is described in the
Installation documentation.
```

The following is a representative subset of the 28 lines of this machine's machine file.

```
ANSI_C 0   # 1 if really ANSI C - ___STDC___ not trustworthy
BOOTCFLAGS -g   # Cflags when booting
CFLAGS -O   # default Cflags when installing
CONTAXDIR %/contax   # default location for the contax database
DESTDIR /usr/dtree   # default location for the target production
DORANLIB 1   # 1 if do ranlib, 0 otherwise
DTREE /usr/dtree   # default location for the installed dtree
INSTFLAGS -qaIs   # default instal(1) flags
INSTLOG @LclVarDir/Ilog   # name of instal(1) audit trail
MISSINGRTNS No_strchr No_strrchr No_mkstemp # missing routines
MUSTALIGN 1   # 1 if words may only be accessed on word boundaries
OPTIONS   # e.g., WHITEPAGES,
ORGANIZATION Nixdorf Computer Canada Ltd.
PATH /bin:/usr/bin:/usr/ucb   # default user path
SYSTEM pyramid_4.2bsd   # brand and version
```

Note that some of the above values could be deduced by programs, but such processes can go wrong and when they do they are difficult to correct. The SetUp process creates a configuration file that is the Machine file plus the source directory name and the name of the machine file itself. This is copied into the ObjDir tree, which was also created by SetUp. The first phase of the construction process is to use the variable value pairs to process prototype header files. For example, the following is the prototype header file that contains the default Dtree location:

```
#ifndef  DTREE
#define  DTREE    "@DTREE@"
#endif   /* not DTREE */
```

The "@DTREE@" is replaced by the DTREE value as assigned in the configuration file.

**References**

1 T.Lord, *Polices and Tools for Hierarchically Managed Source Code Development*, San Francisco, 1988.

2 D.M.Tilbrook and Z.Stern, *Cleaning Up UNIX Source -or- Bringing Discipline to Anarchy.*, EUUG Dublin Conference Proceedings, September 1987.

3 D.M.Tilbrook and P.R.H.Place, *Tools for the Maintenance and Installation of a Large Software Distribution*, EUUG Florence Conference Proceedings, April 1986. Usenix Atlanta Conference Proceedings, June 1986.

4 D.M.Tilbrook, D.Thompson, and M.Lorence, *A New Look at Some Old Problems*, Unix Review, April, 1988.

5 L.Branagan and D.M.Tilbrook, *Installation Documentation Documentation*, EUUG Portugal Conference, Oct. 1988.

6 D.M.Tilbrook, *Quod Erat Faciendum*, unpublished.

*NOTES*